ROBOT ASSISTED MATERIAL HANDLING
FOR SHIRT COLLAR MANUFACTURING
-- TURNING AND PRESSING --
DLA 900-87-0017 Task 0004

FINAL REPORT

VOLUME V:

Three-Dimensional Machine Vision

CENTER FOR ADVANCED MANUFACTURING

CLEMSON
UNIVERSITY
College of Engineering
Clemson, South Carolina 29634

ROBOT ASSISTED MATERIAL HANDLING FOR SHIRT COLLAR MANUFACTURING -- TURNING AND PRESSING --

DLA 900-87-0017 Task 0004

FINAL REPORT

VOLUME V:

Three-Dimensional Machine Vision

Frank W. Paul
Principal Investigator

and

David R. Cultice
Research Assistant

Center for Advanced Manufacturing
and
Clemson Apparel Research

Clemson University
Clemson, SC

June 1992

ACKNOWLEDGMENTS

The authors would like to thank all those who were involved with support of this project. Especially, it is appropriate to thank the Defense Logistics Agency, Department of Defense, for support of this work under contract number DLA 900-87-0017 Task 0004. This work was conducted through Clemson Apparel Research, a facility whose purpose is the advancement of apparel manufacturing technology and by the Center for Advanced Manufacturing, Clemson University.

ABSTRACT

Apparel manufacturers are interested in applying advanced automation techniques to gain increased productivity for existing labor-intensive garment assembly processes. The objective of this thesis is to investigate the usefulness of three-dimensional vision sensing for fabric material manipulation. A three-dimensional (3-D) range finder composed of a charge coupled device (CCD) camera and a facility for projecting laser stripes has been developed and implemented in a robotic workstation dedicated for turning and pressing shirt collars. Through image processing and triangulation, the range finder develops position information which is used by the robot to properly position the collar on a pressing surface.

The scanning and modeling of a collar takes between 16 and 18 seconds, and the complete sensing and positioning procedure requires 20 seconds with robot velocities of 20 ips. The range finder is capable of locating a 3-D point in space to within a ±2 mm precision in the depth direction, and to within ±1 mm in the orthonormal directions. Collars are positioned on the pressing work surface with a precision of ±4 mm which is within tolerance for a successful collar placement. This application of 3-D vision sensing for flexible fabric manipulation demonstrates the promising potential of 3-D vision sensing for apparel manufacturing.

TABLE OF CONTENTS

Table of Contents (Continued)

LIST OF TABLES

LIST OF FIGURES

List of Figures (Continued)

List of Figures (Continued)

CHAPTER I

INTRODUCTION

Clothing manufacturers in the United States face a difficult financial situation due to the economic pressure to compete with low-cost imports. Since clothing manufacture is labor intensive, retail companies generally choose to import assembled apparel from countries with low labor costs. This has reduced prices and profit margins for domestic clothing manufacturers and has created an employment depression within the U.S. apparel industry. Retailers also demand a quick response along with greater flexibility and variety in design from the manufacturer. The development of flexible clothing automation can lower manufacturing costs for a garment manufacturer through enhanced productivity. Automation is essential if the domestic apparel manufacturing industry [1] is to survive in the world market; and many major clothing automation research programs are in progress or have been completed in Japan, Europe, and in the United States.

Robotics, machine vision, and other sensors such as proximity and force sensors are used by manufacturers in the automotive, pharmaceutical, and electronic industries. However, due primarily to the high cost of required research and capital investment, the apparel industry has lagged dangerously behind in applying advanced automation. Transferring automation techniques from other industries to the

apparel industry is not a simple task since apparel fabrics are inherently limp, whereas rigid objects are processed in other manufacturing industries. Due to imprecise structural geometry, processing limp fabrics requires an added magnitude of research, ordinarily unwarranted when automating a process in other industries. Thus, new handling and sensing techniques are necessary to facilitate an automated apparel process.

This thesis addresses the use of three-dimensional (3-D) vision for sensing limp fabric material. Vision sensing is a complementary technology to automating processes, permitting visual process monitoring, often in real-time. Automated apparel processes stand to benefit greatly from the implementation of 3-D visual sensors.

Background

The present state-of-the-art in advanced commercial sewing equipment uses semi-automatic sewing units which reduce the requirement for a skilled operator [2]. These units are characterized by high specialization, minimal programmability, and limited flexibility to style changes; they require frequent manual adjustments to accommodate different sizes and fabric types. These machines are best described as semi-hard automation devices. Handling operations such as ply separation, parts mating, and parts feeding are often currently performed manually [3].

An automated sewing system requires a comprehensive and adaptable sensing capability to detect the location and

condition of fabric workpieces. Though many sensing technologies are available and have been successfully employed in the past, machine vision is an excellent method for geometric sensing because of its non-contact nature. Norton-Wayne [4] points out that the eye-brain system of the human worker is required at many stages of the apparel manufacturing process, and can be replaced by the machine vision functions of pattern recognition and image analysis to aid in achieving automation. Reluctance by the apparel industry to employ machine vision has been a result of high cost and the misconception that machine vision is unreliable and requires complicated programming. However, due to lower hardware costs and improved technologies, machine vision is beginning to gain acceptance in the industry as exemplified by the several sewing units, currently on the market, which utilize optical sensors for real-time edge tracking and locating fabric plies for alignment.

A research team at Clemson University investigating advanced apparel automation has undertaken a project to automate an existing shirt collar processing task. A shirt collar is composed of two plies of shirt fabric and an inner liner. The plies are aligned with the shirt fabric together and the liner on top and then stitched 3/16 inch from the ply border on three sides. The unsewn border enables the collar to be turned such that the liner is on the inside and the rough edges are hidden as shown in Figure 1.1. Currently, one of the most labor intensive tasks within the apparel

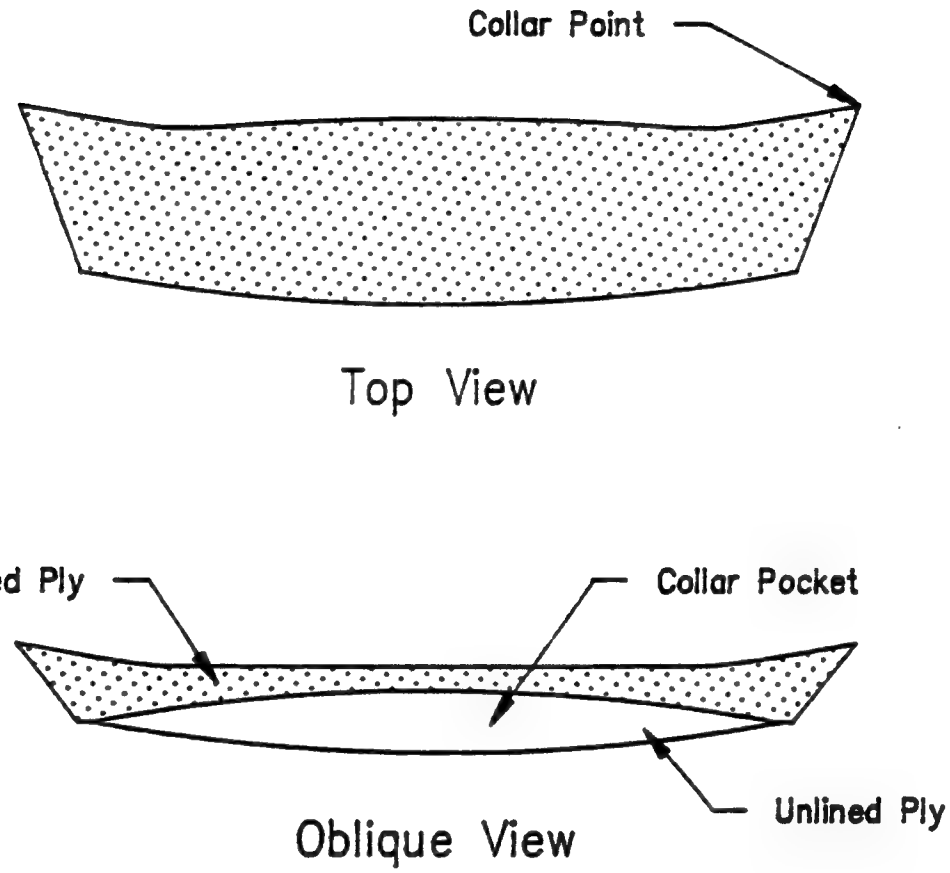


Figure 1.1. Illustration of turned shirt collar.

industry is the process of turning a shirt collar right side out, and pressing it with its edge seam aligned directly along the collar perimeter. In today's factories, an operator uses a manually operated machine to aid in inverting and pressing shirt collars.

The operator's responsibilities are primarily hand-eye coordination tasks which involve locating the collar, applying tension to the collar, and aligning the collar seam along the edge of a creaser blade. It is critical to (1) create sharp collar points and (2) align the seam along the outer edge of the pressed collar for the collar to be accepted.

A workstation has been developed with an AdeptOne SCARA robot and two dedicated in-house designed machines: one to invert or "turn" the collar, and the other to align the collar seam and press the collar to automate the shirt collar turning and pressing process. Figure 1.2 shows the schematic layout for the workstation. The robot maneuvers the collar between the process stages with a robotic handler, an end-effector designed for collar manipulation [22]. The collar must be retrieved from the turning machine, transported to the pressing machine, and positioned in a precise location at the pressing station. The collar positioning task involves vision sensing, since the collar is not rigid. Uncertainty is introduced in the position of the collar relative to the end-effector when the robot retrieves the collar from the turning machine. A three dimensional machine vision process

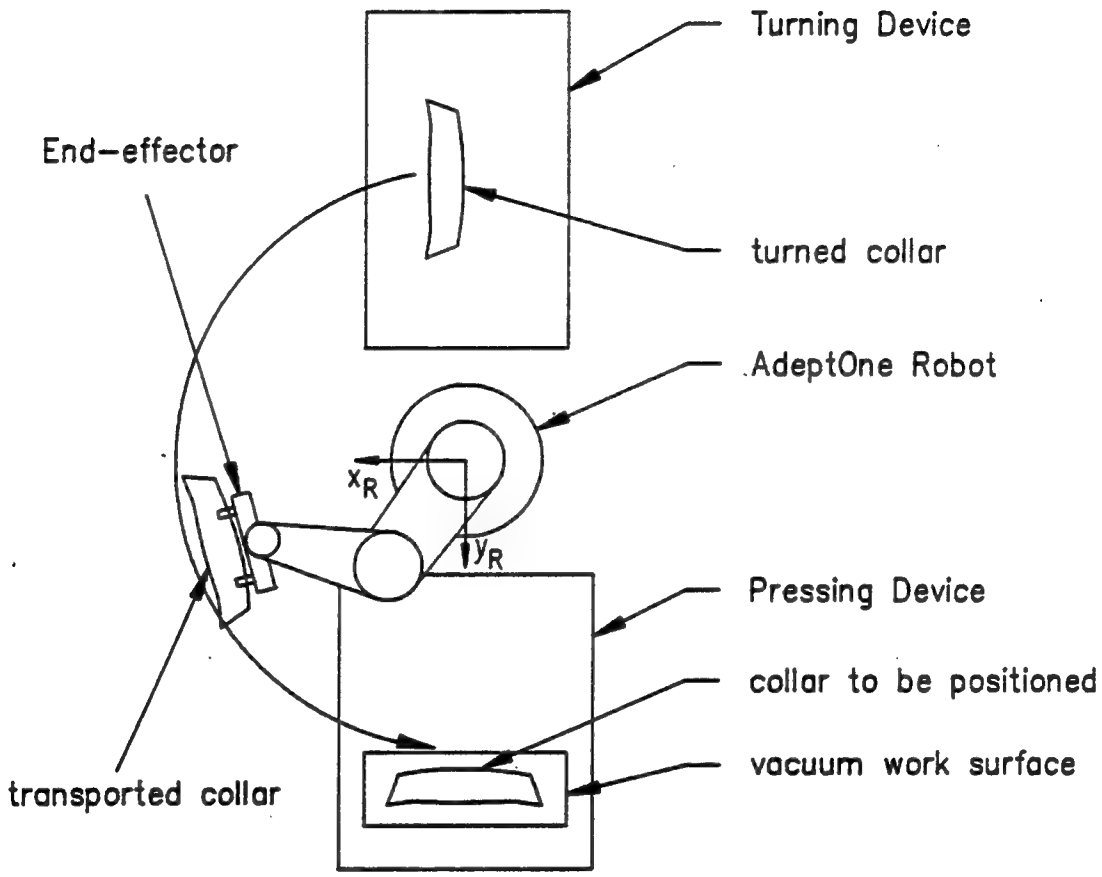


Figure 1.2. Research workstation layout (overhead view).

is used to identify the collar position relative to the end-effector for placement at the pressing station.

Literature Survey

The literature reports that a significant amount of research has been conducted in the fields of machine vision and image processing. In this thesis, the literature which addresses the apparel is divided into three general groupings: (1) research and applications of two-dimensional vision, (2) research and applications of three-dimensional vision, and (3) general 3-D vision research with strong possibilities for application in the apparel industry.

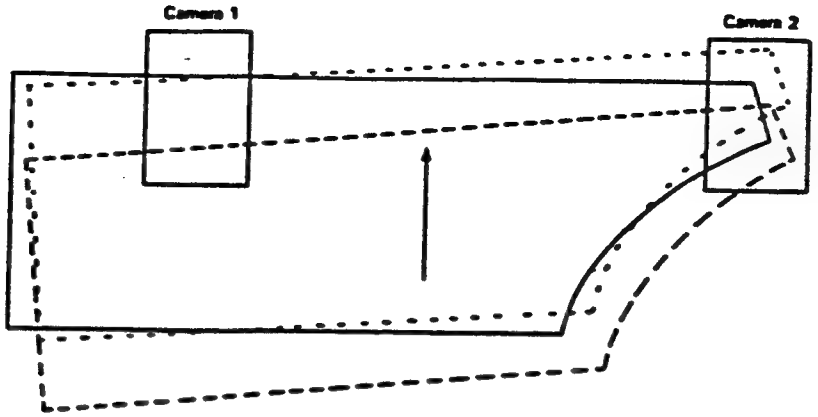
Two-Dimensional Vision Research in the Apparel Industry

Two-dimensional machine vision involves observing planar surfaces which are normal to the camera line of sight. A simple calibration conversion between CCD camera (charge coupled device) pixels and the viewed surface area permits evaluation of the image. Due to its relative simplicity, and perhaps greater application demand, two-dimensional vision applications are more prevalent in the apparel industry than three-dimensional applications. Gershon and Porat's vision servo controlled sewing system directs the manipulation of fabric panels beneath a sewing head [2]. A camera is used to locate the position of the fabric ply, and with this positional information the system is able to respond with a servoed action. Similarly, Taylor's [5] work has involved matching plies which are to be joined; Kelley [6] has worked

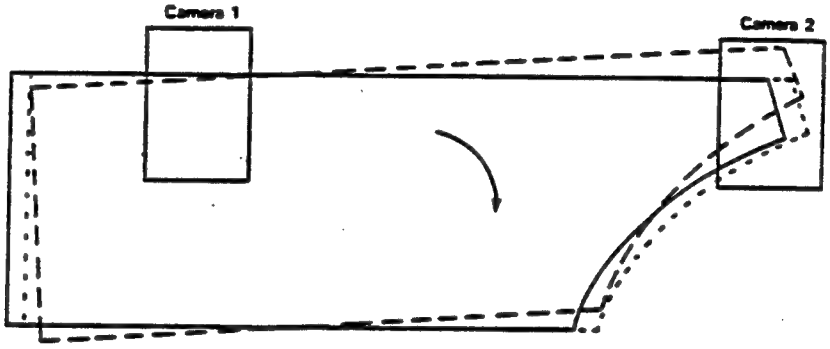
with the sensing and picking of plies from stacks and conveyor belts; and Iype and Porat's [7] work has included fabric alignment. Figure 1.3 displays the process of utilizing two cameras for fabric alignment as described in [7]. Torgerson and Paul's vision guided robot system analyzes the profile of a fabric panel and deduces the manipulator path to produce an approximate edge seam equidistant from the workpiece border [8].

Three-Dimensional Vision Research in the Apparel Industry

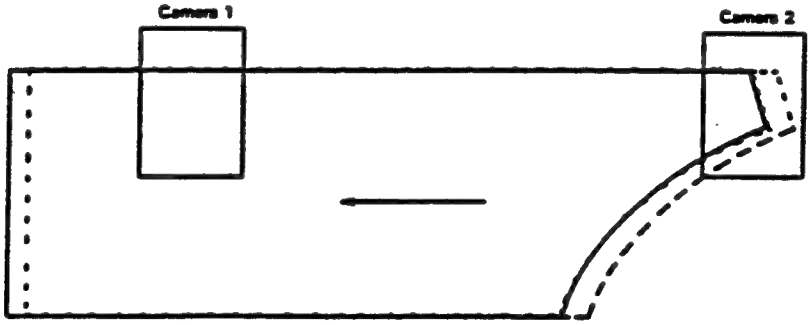
Three-dimensional vision sensing of flexible materials is an area which has extensive possibilities in the apparel industry. Apparel workers routinely use their sense of vision in material manipulation, suggesting that 3-D visual sensing is important in the automation of apparel fabric handling and processing. Although some 3-D processes can be simplified to 2-D tasks for automation purposes, others cannot be changed and require the use of 3-D visual sensing. Geometric measurement and inspection of semifinished or finished garments are possible uses of 3-D machine vision. Another use involves detecting an object in 3-D space, for the purpose of sewing by a 3-D robotic sewing head. Juki (Japan) has demonstrated a robot with a sewing head for joining 3-D seams, such as where the seam joins the sleeve and shoulder workpieces [9]. Taylor [3] states that 3-D manipulations may involve folding fabric subassemblies,



(a) *X* and *Y* Movement for Camera 1.



(b) Rotational Movement for Camera 1



(c) *X* and *Y* Movement for Camera 2

- — — original position of the fabric.
- —— position of the fabric before move.
- —— position of the fabric after move.

Figure 1.3. Fabric alignment with two cameras [7].

turning garments inside out (inversion), or sewing along curved edges.

Halioua [10] has demonstrated the phase-measuring profilometry method for sensing the surface of the human body to aid in the design and fitting of garments. This method involves projecting a sinusoidal grating structure onto the surface to be measured, and detecting the resulting deformed grating image at an offset angle by a CCD-array camera. Figure 1.4 demonstrates an example of such a grating on a human body. A microprocessor processes the images using interferomic phase-measuring algorithms to arrive at a graphics model of the image. The graphics model is then converted into fabric cut-size patterns.

Other Three-dimensional Vision Activity

Sato et al. [11,12] has developed a 3-D surface measurement system utilizing a liquid crystal mask which by masking a projector creates a predefined pattern on the unknown surface. A short series of these computer commanded patterns is projected on an object, and the acquired camera images of the resulting deformed patterns are analyzed for depth information commonly known as range data or range information. Figure 1.5 shows an object scanned by Sato's projector with nematic crystal mask technology. The recovered range information is used in automation to interact with the located object, either by manipulating it, performing an action on it, or possibly avoiding collision with it.

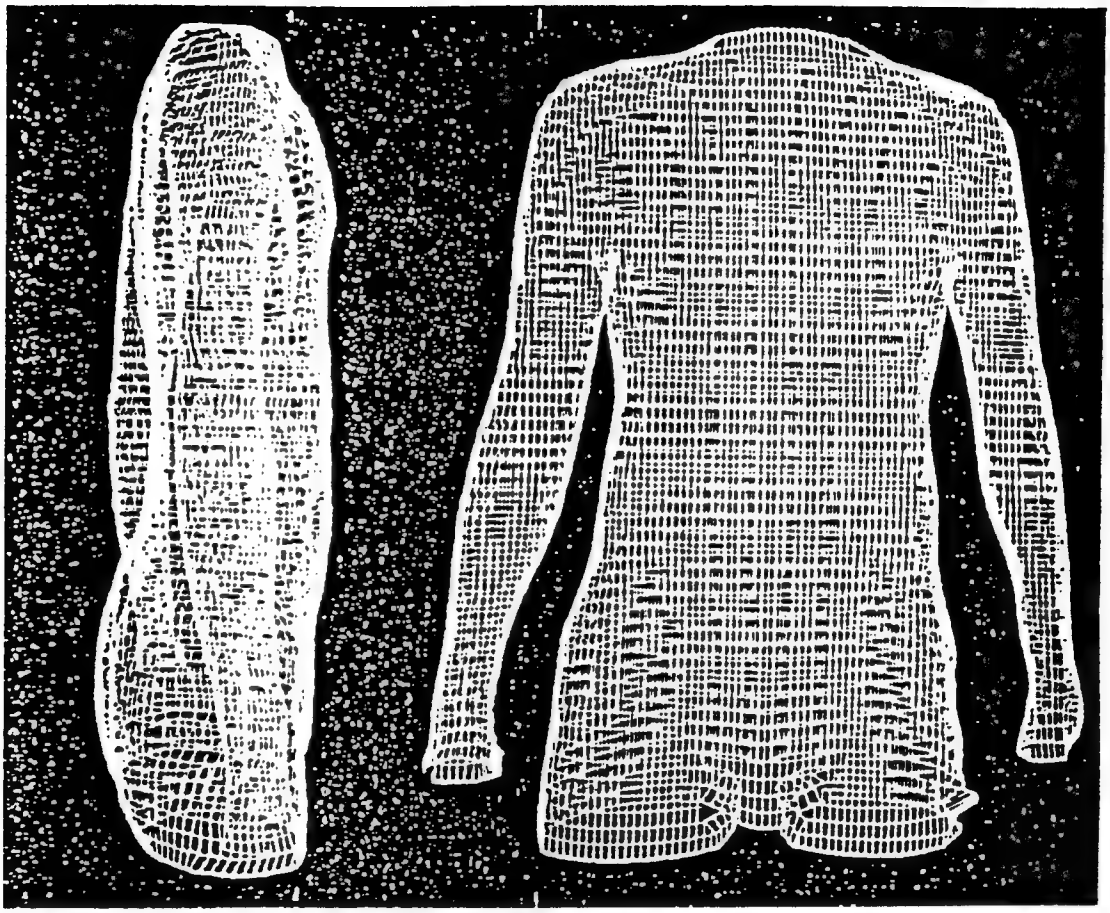
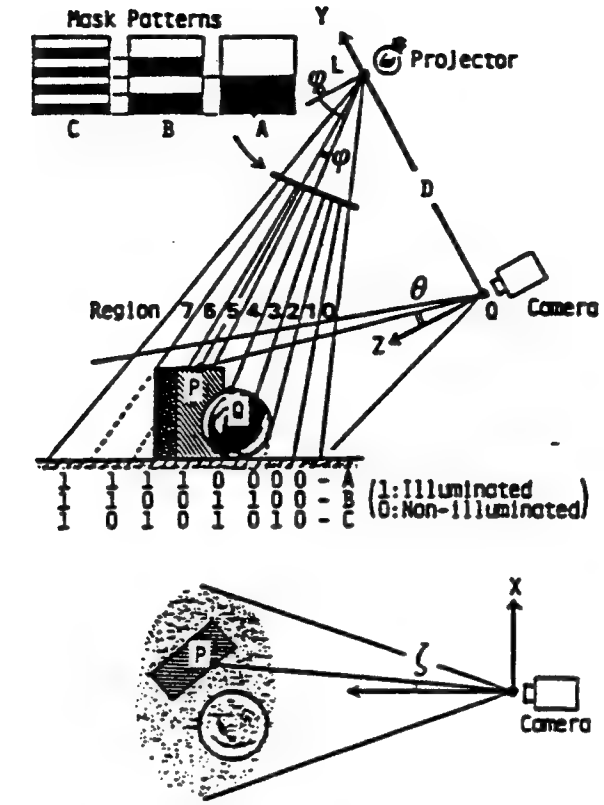


Figure 1.4. Profilometry for automated garment design [10].



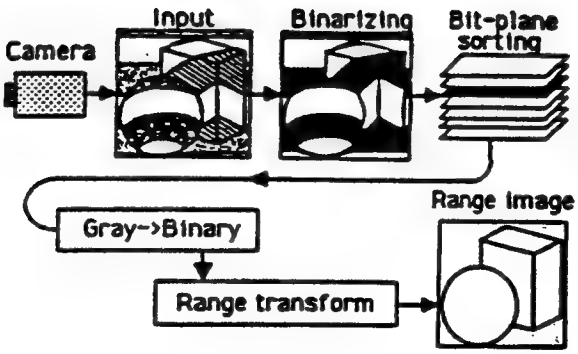
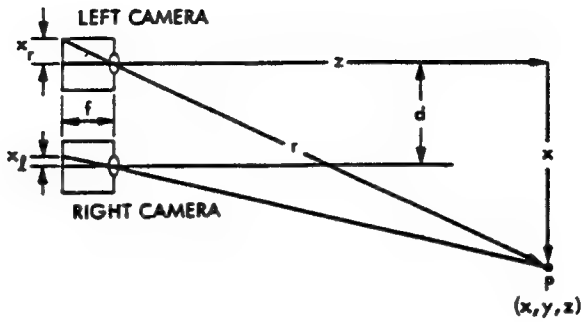


Figure 1.5. Principle of range finding by space encoding [11,12].

Three-dimensional range imaging is an established field of research and application [13]. Applications include inspection and geometric measurement of conveyed or still items; sensing of contours for planned path control for 3-D welding, gluing, or seam sewing operations; and sensing of the environment for controlling the movement of an automated guided vehicle (AGV) or robot. Computed tomography (CT), used to sense the internal organs of the human body, CAT scanning, and ultrasound imaging are other familiar forms of 3-D range sensing [14]. As computer processing speeds and memory capability improve, 3-D vision systems will become less expensive and more accepted for many complex industrial applications.

Many of the 3-D sensing methods available use triangulation to extract 3-D data. This is typically accomplished in one of two ways: (1) with stereo cameras or (2) with one camera and structured lighting [15]. Structured lighting is defined as a calibrated projected grid of line(s) or dot(s) from a light source. Typically, the light source is a projector with a grating lens or a laser with appropriate optics. The displayed pattern on the object is evaluated by the single camera to determine range information. Stereo vision involves the relative position of individual images of common points in a camera pair, as in Figure 1.6. These common points are used with triangulation to determine range data. Structured lighting does not involve recognizing such points since the machine vision algorithm need only detect



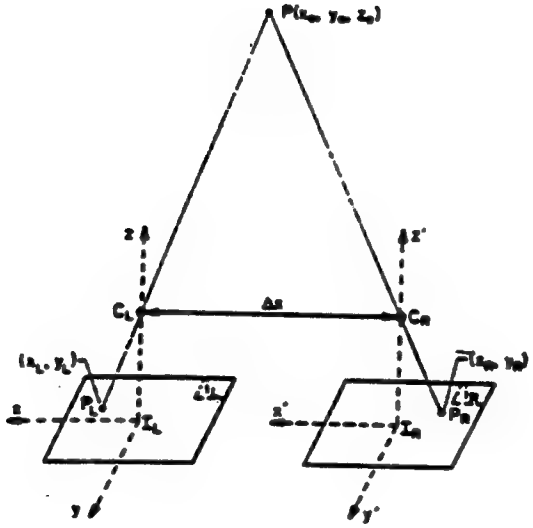


Figure 1.6 A stereo camera model for range finding [20].

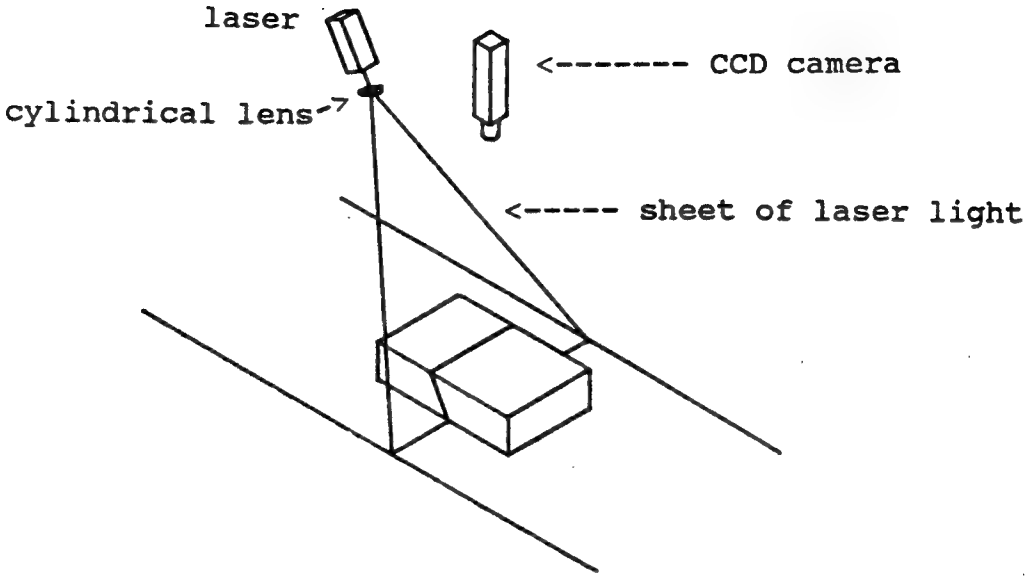
the variations of the projected pattern on the object, as shown in Figure 1.7. Structured light, particularly from a laser source, offers the added benefit of being easily distinguished from ambient or factory lighting. This research uses structured laser light in conjunction with a CCD camera to perform 3-D sensing.

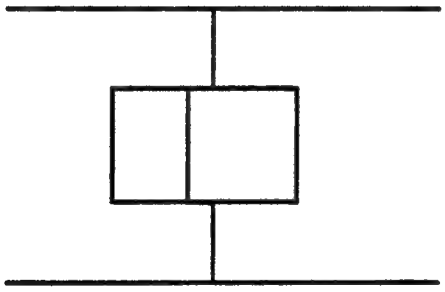
Research Objectives and Problem Statement

The research objective of this thesis is to contribute to the research base necessary to make 3-D vision sensing useful for the apparel industry. The ability to determine the surface profile and object geometry of flexible fabric workpieces is required for many automated handling and manipulation applications since the workpiece shape and position can vary. Typical workpieces range from single ply materials to semifinished subassemblies such as a sleeve or collar from a shirt.

To demonstrate the use of three-dimensional vision sensing, an application has been selected which involves positioning a shirt collar within the workspace of an apparel assembly workstation. It is critical to position the collar to its specified location with precision. The problem statement can be formally stated as:

> Develop a system to position a shirt collar within a toleranced location on a work surface. A three-dimensional vision sensor is used to provide the required feedback, and thus this work will provide an investigation of three-dimensional vision sensing for apparel applications.





laser stripe imaged by camera

Figure 1.7. A structured light example.

To perform 3-D vision sensing, considerations must be given to: (1) the hardware to generate a single sheet of laser light at variable positions, (2) the calibrated geometry for determining range data, and (3) image processing techniques for detecting the light stripe on the acquired images. A mathematical collar model is developed to describe the collar geometry prior to loading on a work surface. The model is compared with a desired collar location on the work surface to produce an error vector of rotation and translation components. The error vector is used to generate a robot motion trajectory to position the collar acceptably on the work surface.

Thesis Organization

Three-dimensional vision sensing is an important method for locating and positioning fabric and garment assemblies. This thesis will present an apparel application involving 3-D sensing. Chapter II discusses the fundamentals in locating and positioning the collar on the work surface along with issues of 3-D vision sensing. Chapter III describes the hardware equipment and software algorithms used to accomplish the collar locating and positioning procedures. Chapter IV provides an analysis and discussion of the system performance while Chapter V presents conclusions and recommendations derived from this work.

CHAPTER II

VISION SENSING FOR LIMP MATERIAL HANDLING

Problem Structure

A robot with end-effector is used in the apparel assembly workstation (AAW) to acquire and move a newly turned shirt collar from the turning device to the work surface of the pressing device. The pressing device acts to insert a set of creaser blades into the collar opening, with the blades aligned inside the turned collar pocket. For this reason, it is critical to accurately position the collar (the collar points are used for reference) on the creaser blades.

The position of the collar is unknown relative to the end-effector grippers. However, since the collar is acquired from the turning machine, a bounded region exists for the collar location relative to the end-effector. Given the three-dimensional geometry and inherent limp material characteristics of a collar, 3-D vision is employed to locate the position of the collar as grasped by the end-effector.

When the collar is held by the end-effector, it has a pouch-like geometry, with the unstiffened ply grasped by the end-effector and the stiffened ply draped to create an open pocket as depicted in Figure 2.1. Since the top ply fabric is limp, the 3-D collar geometry is altered when in contact with the work surface of the pressing station.

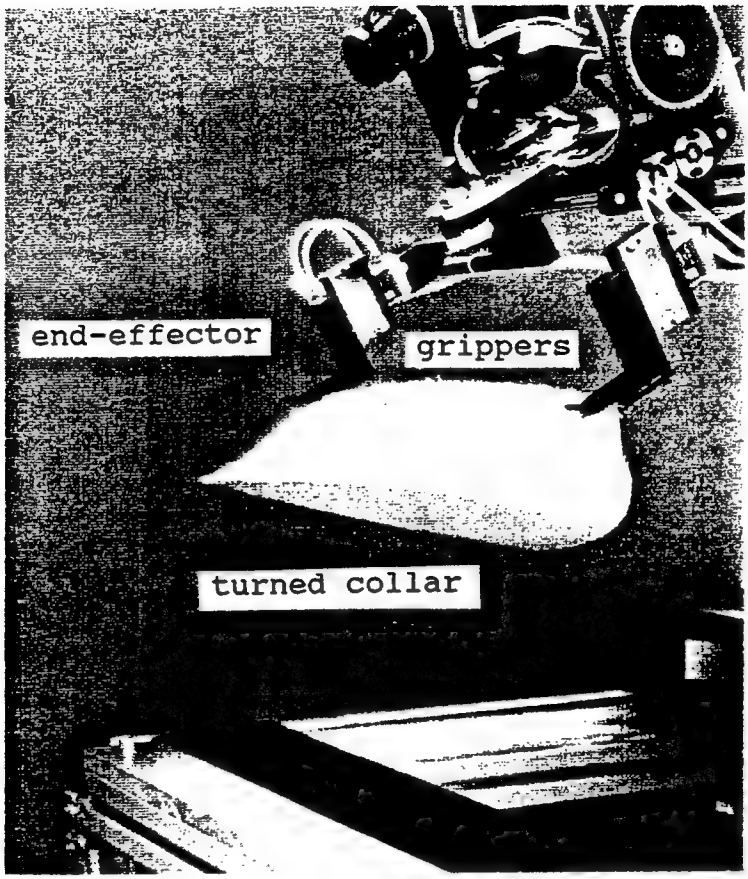


Figure 2.1. End-effector with turned collar.

A turned collar resting on the work surface is similar to a two dimensional object, suggesting that a two dimensional means of sensing might be used to position the collar. The end-effector, however, is incapable of guiding the collar to instructed locations on a surface. This dilemma is inherent to the geometry and material properties of the grasped collar. Since the opened collar is not a rigid object and friction exists between the lower ply and the work surface, the lower ply will not move identically with the robot. Instead, the top ply shears relative to the lower ply, with the geometric relationship between upper and lower plies difficult to predict.

The selected method for maneuvering the collar involves adjusting its location in the space directly over the work surface, prior to placing it on the surface. This allows each collar to be loaded on the surface identically, without shearing motions against the work surface. A vacuum is drawn through the surface immediately following collar contact with the surface. This helps hold the collar in position prior to insertion of the pressing creaser blades.

A mathematical model of the collar has been developed which analytically defines the position of a collar hanging from the end-effector prior to loading. This model is used to identify the location of the collar points in 3-D space and their location relative to the end-effector. With this knowledge and several properties inherent to a turned collar,

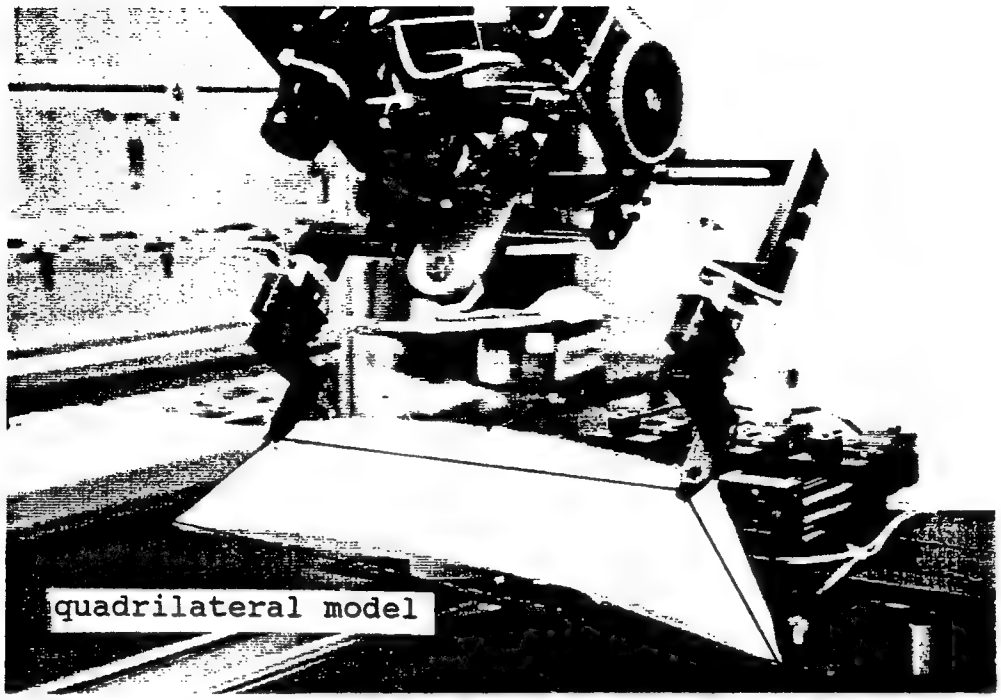
it is possible to predict the future location of the collar points when placed on the flat work surface.

A turned collar held by the end-effector is pouch-like as previously shown in Figure 2.1. Both plies, including the stiffening liner, are folded along the seam creating a relatively stiff connection between the two collar points, prohibiting the seam to droop or crease. The seam stiffness, coupled with the collar point locations and the end-effector gripper positions, are the basis for the mathematical collar model.

The Collar Model

The end-effector gripper positions are given by the forward kinematic equations involving the robot position and the particular end-effector configuration. The AdeptOne robot features good repeatability due to its direct drive SCARA design. Thus, it is possible to locate the collar points relative to the end-effector grippers, implying that the points are known relative to the robot position. As a result, the collar location is defined within the AAW workspace.

A quadrilateral model of the collar, as shown in Figure 2.2, has been developed which spatially relates the collar points to the end-effector grippers. The wireframe model is defined by the two collar point positions and the two points of contact made between the ends of the grippers and the top ply fabric. The quadrilateral, which defines the top fabric ply (unlined), retains a constant geometry as long as uniform



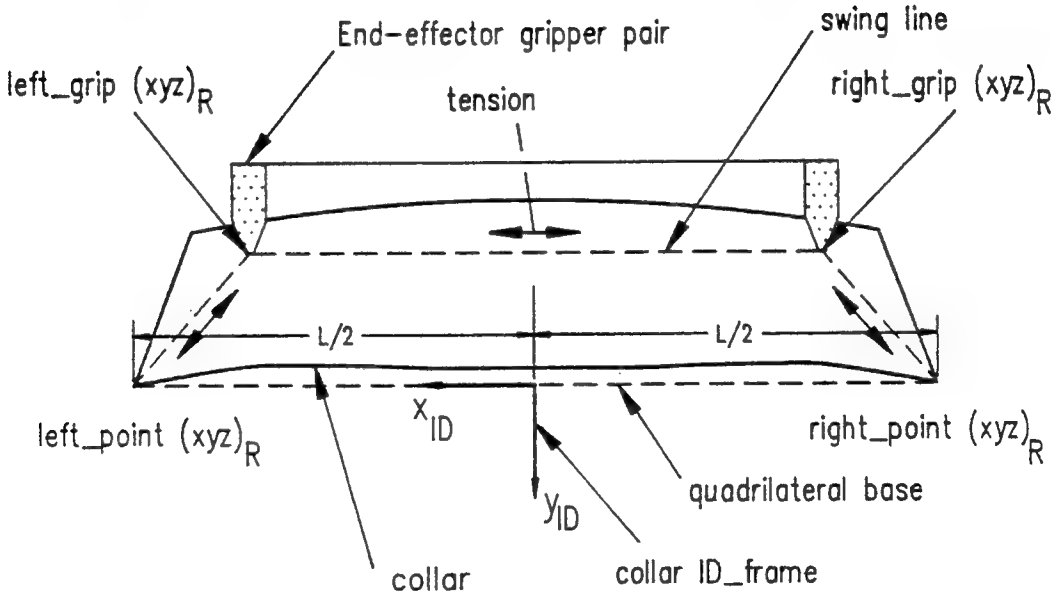


Figure 2.2. The quadrilateral collar model.

tension exists between collar points and grippers. This is the case when the collar hangs freely from the end-effector. A slight tension must be maintained as the collar is lowered onto the vacuum surface.

The model predicts the location of the collar points as they are positioned on the vacuum surface. The collar swings freely from the end-effector about an axis of rotation defined between the tips of the grippers prior to loading. Figure 2.3 illustrates the end-effector grippers holding a collar and demonstrates the "swing line". The swing line enables the ungrasped collar geometry to remain unchanged regardless of the end-effector pitch angle, as shown in Figure 2.4. Assuming that the quadrilateral geometry is known, the collar can be positioned on the vacuum surface by commanded robot motions designed to align the collar into the desired workstation position. A tension applied by the end-effector perpendicular to the base of the quadrilateral, is required to keep the top ply tensioned and the quadrilateral geometry intact after the collar contacts the work surface.

Three dimensional vision sensing is responsible for evaluating the location of the collar points in 3-D space, and the end-effector gripper tip locations are determined through forward kinematics. It is desirable to adjust the position of the collar prior to surface contact as though the collar were in a plane parallel to the work surface. This method for positioning is used for two reasons: (1) the work surface plane corresponds to the only available robot plane

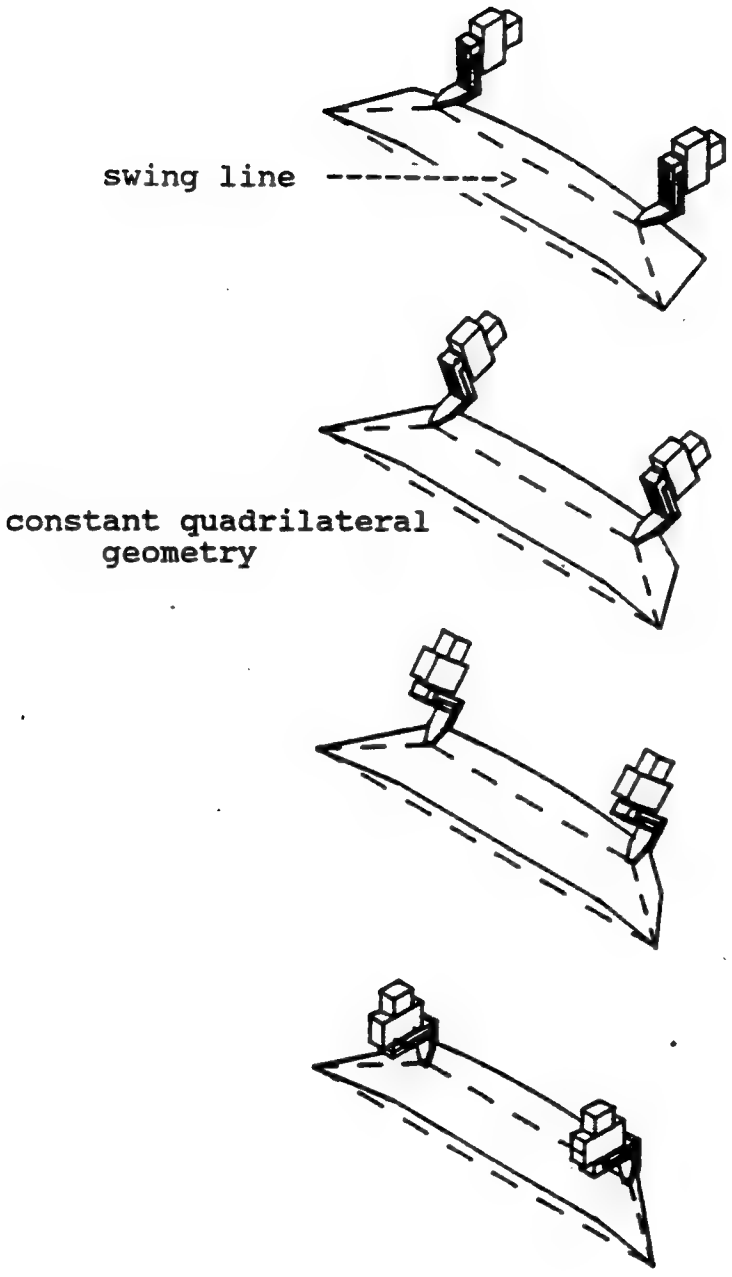


Figure 2.3. Demonstration of collar swing line.

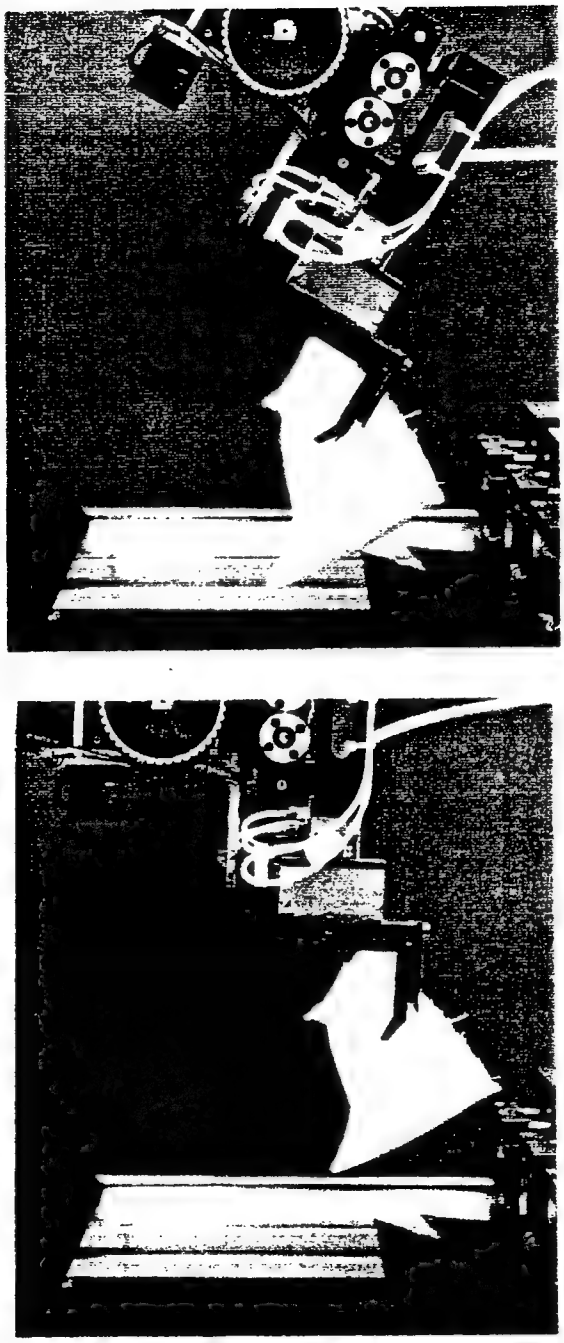


Figure 2.4. Further demonstration of collar swing line.

of rotation, and (2) the collar must be oriented in this plane prior to surface contact due to friction between bottom ply and work surface. Theoretical collar point positions are calculated for the collar in a plane parallel to the vacuum surface and common to the end-effector gripper tip locations. This is accomplished by rotating the measured 3-D collar point coordinates about the collar axis of rotation until the quadrilateral surface is in the desired parallel plane.

The coordinate frame "ID_frame" is assigned to the model identifying the collar position after the wireframe collar model is established between the gripper tips and the collar points in this plane. Figure 2.5 shows the hypothetical case of a collar parallel to the vacuum surface. The coordinate frame ID_frame is located equidistant from the collar points, as shown previously in Figure 2.2, with the x-axis along the base of the quadrilateral, and the y-axis perpendicular to the base directed away from the quadrilateral.

A coordinate frame "target_frame", shown in Figure 2.6, is located equidistant between the creaser blade tip locations after they are inserted and extended in the collar. The outlined contour on the vacuum surface indicates the inserted and extended creaser blade positions, which coincide with the loaded collar position. The collar coordinate frames, ID_frame and target_frame, provide for a position match between the collar to be loaded and a predetermined collar position on the vacuum surface.

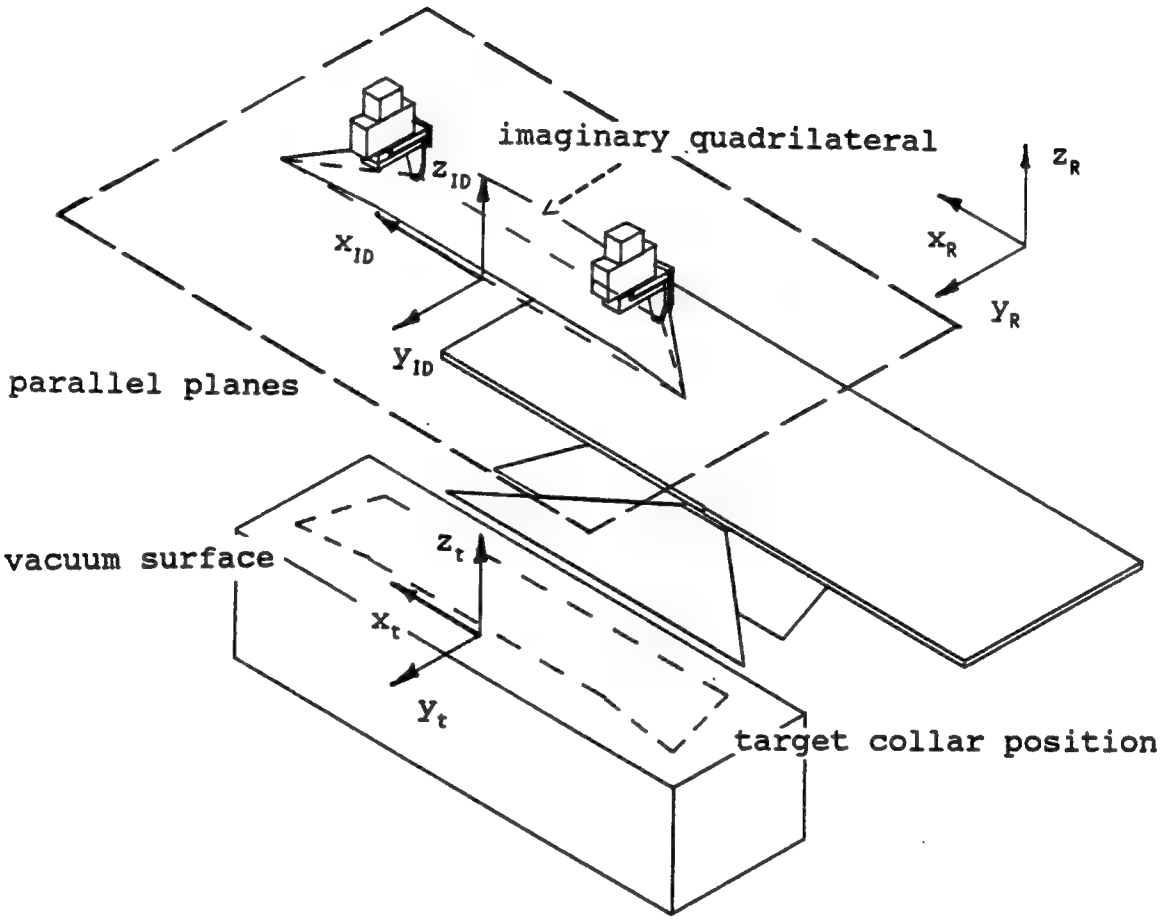


Figure 2.5. Imaginary quadrilateral plane parallel to vacuum surface.

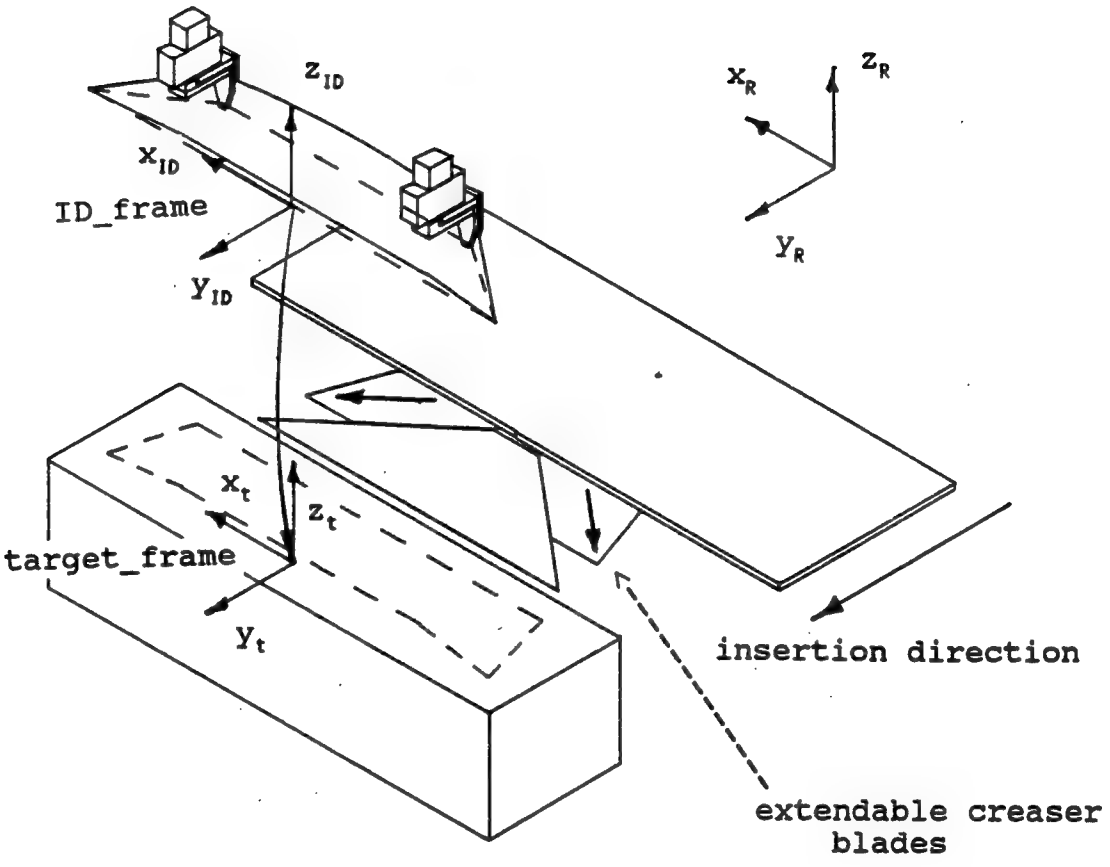
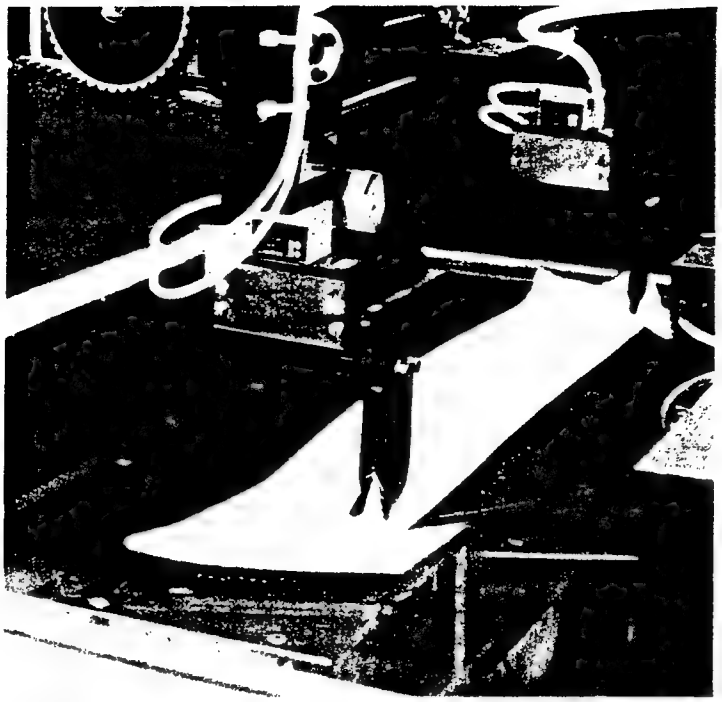


Figure 2.6. ID_frame and target_frame assignments.

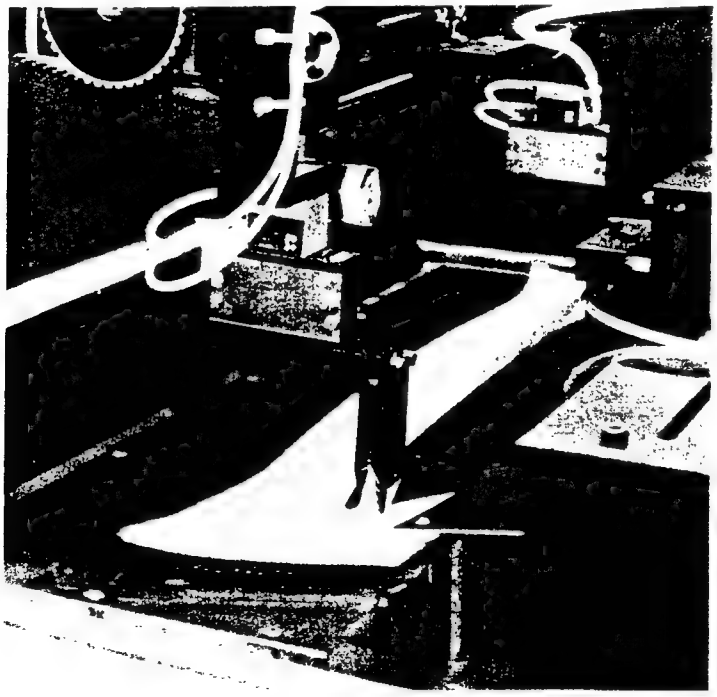
Collar Loading Procedure

The robot retrieves the collar from the turning device and moves to predetermined coordinates where the collar is presented to the pressing area. At this position, the end-effector gripper tip locations define a plane parallel to the pressing surface in which the collar model is evaluated. Coordinate frame ID_frame is assigned to the model with the frame origin in robot coordinates. The ID_frame is compared with the target_frame, also with origin in robot coordinates, to yield an error vector composed of Δx_R and Δy_R translations and an x_R-y_R in-plane rotation about z_t. These coordinates are defined as the difference in ID_frame and target_frame frames with respect to the robot coordinate system. The difference in the z_R direction is constant for all collars since the target_frame location is invariant and the gripper tip locations, which designate the x_{ID}-y_{ID} plane in robot coordinates, are dependent on the predetermined robot position.

A robot trajectory is designed with the error vector information to load the collar in place for creaser blade insertion and extension. Following collar placement by the robot, a clearance remains for the creaser blades to clear the end-effector grippers and top collar ply. The required clearance of 2 mm is shown in Figure 2.7. Additionally, a loaded collar which has been centered on the target_frame y_t axis provides a clearance of 11 mm for each side of the collar pocket between the opening of the collar and the retracted creaser blade tips. Thus, a conservative tolerance



(a) prior to creaser blade insertion



(b) during creaser blade insertion

Figure 2.7 Clearance between creaser blades and loaded collar opening.

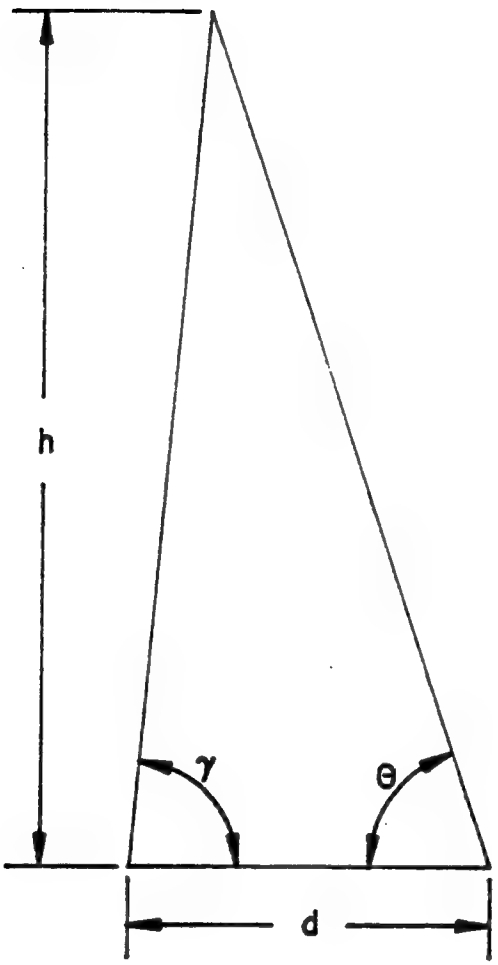
of ±6 mm in the x_t direction is necessary to insure that the creaser blade assembly will insert into the collar pocket. This side-to-side clearance is shown in Figure 2.7.

After the collar has been positioned on the vacuum surface, the AAW process continues with the pressing operation by inserting and extending the creaser blades in the collar. Positioning the collar on the vacuum surface concludes the use of three-dimensional vision for the workstation process.

Three-Dimensional Vision Sensing

The collar points, which are defined in a 3-D coordinate system, are located with 3-D vision using range finding. The implemented technique for obtaining depth data, or spatial coordinates, involves triangulation. Triangulation determines the distance between the apex of a triangle and its base; this distance is computed with knowledge of the base length and its two adjoining angles as shown in Figure 2.8.

A laser stripe source and a camera offset from the laser source compose a system used for detecting range data (depth data) by means of triangulation. An object, such as a collar, in view of the camera and intersecting the plane of the projected laser light sheet features a laser stripe on its surface, and when viewed by the camera represents range data. Figure 2.9 shows the system laser striping configuration with geometry identical to that of Figure 2.8. A CCD camera (charge coupled device) features an image plane composed of a 2-D array of light sensitive pixels. Each pixel registers a



governing equation:

$$h = d/(\tan(\pi/2-\theta) + \tan(\pi/2-\gamma))$$

Figure 2.8. Basic triangulation geometry.

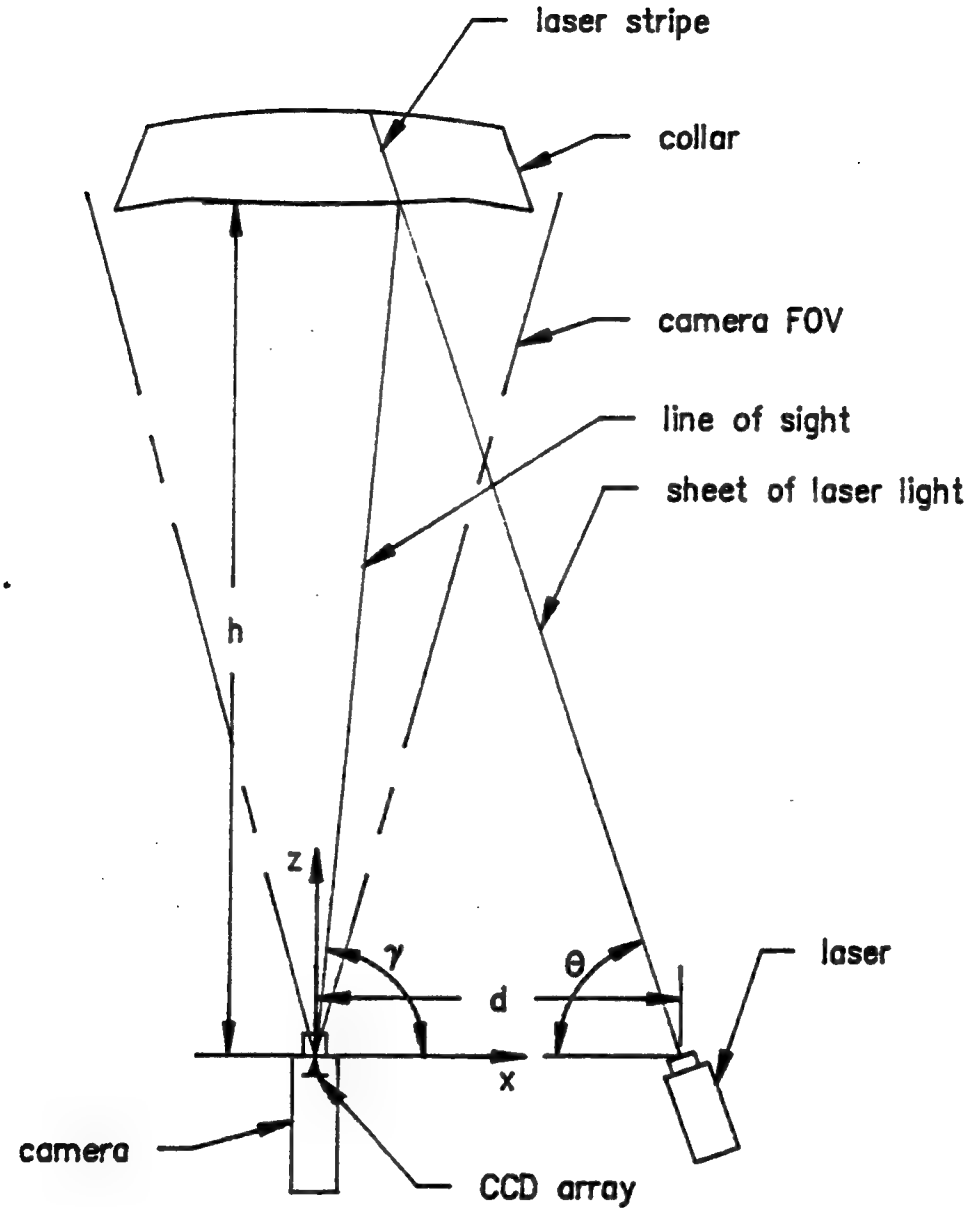


Figure 2.9. Basic laser striping configuration.

brightness intensity value in the vision system software via A/D converters. A digital image, therefore, is a 2-D array of pixel intensity values. The Data Translation IRIS vision system (identified in Appendix A) provides a 512 x 512 pixel array image, with an eight bit intensity resolution (0-255 scale). A CCD camera is an image plane with a lens offset from the image plane by a focal length f. A simple camera configuration known as the pinhole camera model is sketched in Figure 2.10. This model is used for calibrating the camera as well as developing range data equations.

In general, image processing involves processing pixel information from the acquired image as a function of the pixel intensity value. It is crucial to isolate the stripe from the background by filtering and thresholding to evaluate the laser stripe. Once isolated, the stripe can be detected with a pixel scan across the image. Figure 2.11 shows a laser stripe on a collar before and after the image has been processed to isolate the stripe. Each pixel, such as those composing the stripe, has x_i and y_i coordinates defined in the 2-D image plane coordinate system. Range data, or depth data, for any point of the stripe on the collar is determined with the corresponding pixel on the image plane and the respective geometry of the laser plane. Also, the position of the camera and laser plane must be modeled and calibrated to give credence to the pixel information.

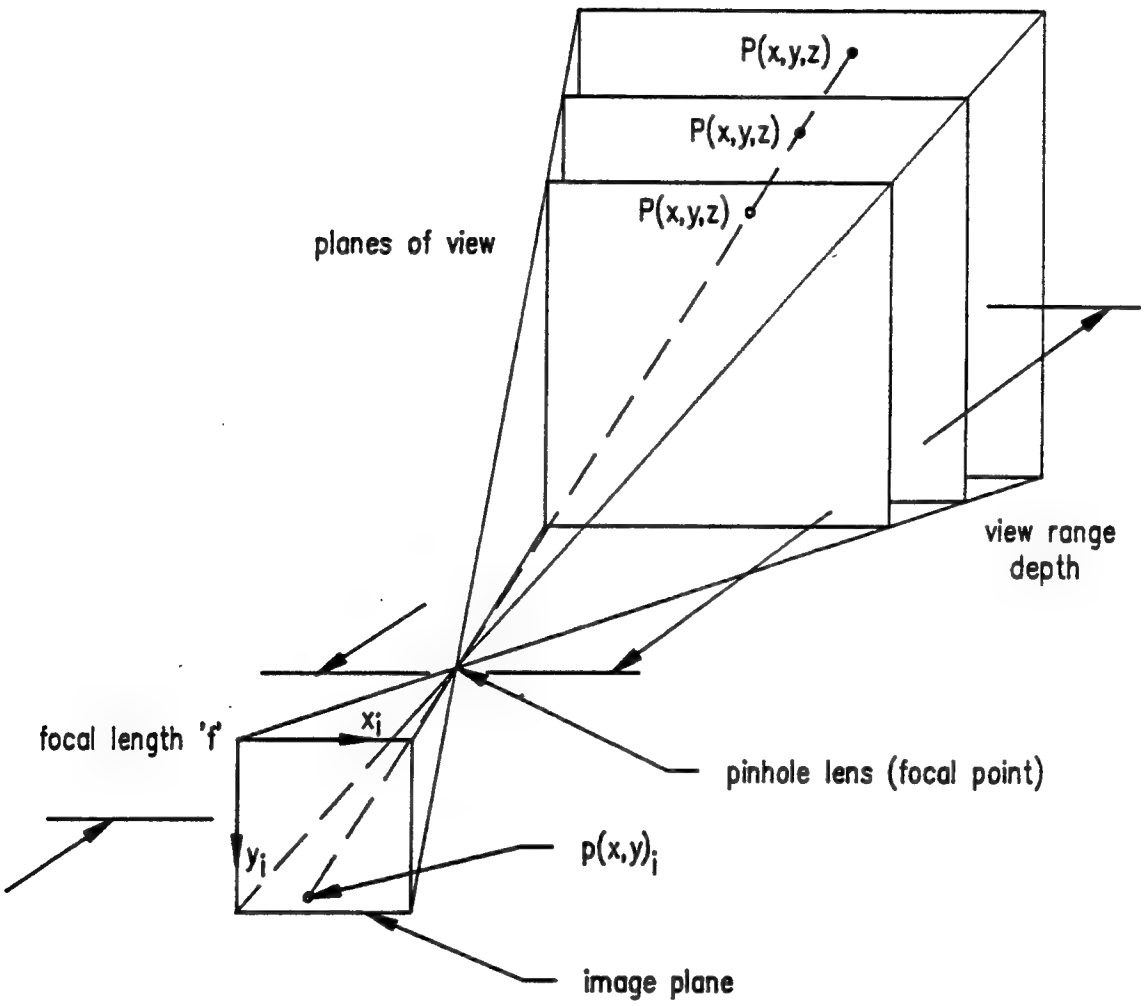
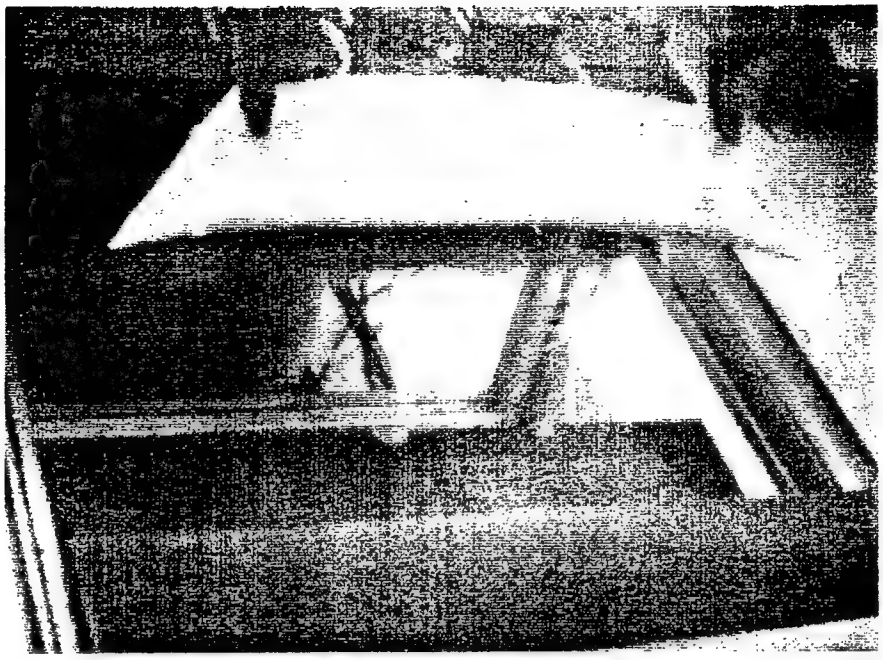
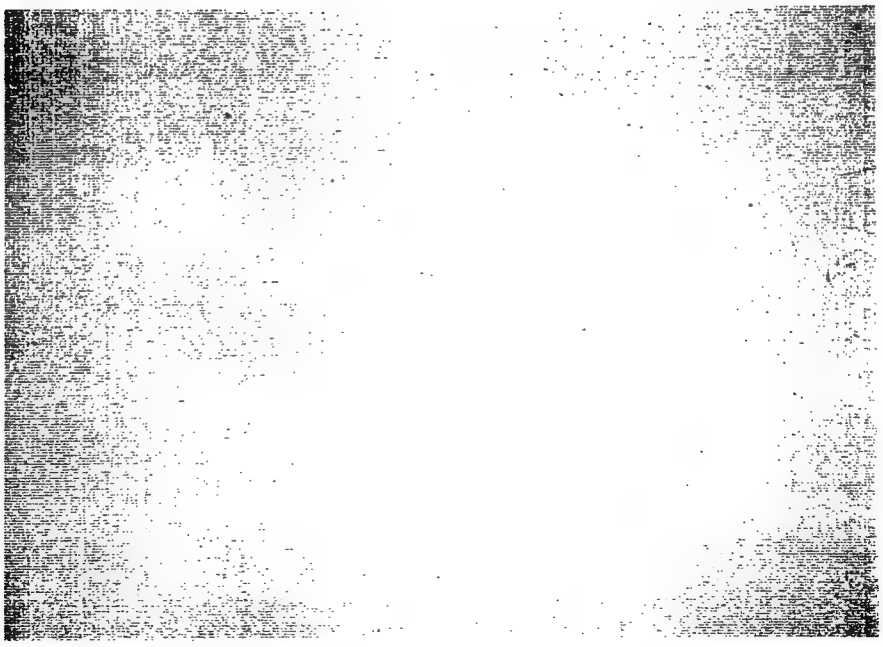


Figure 2.10. General pinhole camera model.



(a) before isolation



(b) after isolation

Figure 2.11. Acquired image of a laser stripe.

Concluding Remarks

A methodology has been developed to measure the location of object points in 3-D space using a CCD camera, HeNe laser, and a rotational mirror for scanning. The device is used to project a sheet of light on the collar, and view the resulting stripe with the camera. Once the vision system has acquired "stripe on collar" image data, image processing techniques isolate and analyze the stripe for the range data. The range information is used to construct a collar model for positioning the collar on the vacuum surface of the pressing device.

Range data measurement accuracy is necessary to locate the collar points and generate a collar model sufficient to successfully load the collar. Accurate range data measurements are a function of the variables involved in triangulation including those in the camera model and the range data instrument geometry. Calibration between the instrument coordinate frame and the robot workspace is important to relate range data to the workstation workspace since both the collar model and the robot trajectories are developed in workstation (robot) coordinates. Additional factors such as robot accuracy and system repeatability are important, but do not contribute significantly to the inaccuracy of the collar model since these factors have significantly higher precision (0.025 mm, 1 mil) as compared to the range data instrumentation (2 mm, 80 mil).

CHAPTER III

VISION SYSTEM IMPLEMENTATION

A vision system range finder has been designed and implemented for an AAW workstation to evaluate the 3-D position of a collar. This chapter details the design and geometry of the range finder instrument named the Range Data Scanner (RDS). A functional description of the RDS follows, including the image processing and computer algorithm necessary to construct the collar model. The interaction between the RDS sensor and AAW workstation is discussed in the context of the system process and its sequence of operation.

Range Data Scanner Design

The RDS has been designed to utilize laser striping and a CCD camera for geometric triangulation. The RDS incorporates a 1 mW HeNe laser in conjunction with a cylindrical lens to generate a vertical sheet of light for striping as shown in Figure 3.1. The lens diffracts the laser beam into a laser sheet of light which is directed at a flat mirror that can be rotated to reflect the light sheet in desired directions. The rotational mirror is mounted to a shaft in a bearing housing and driven by a stepper motor. The laser, mirror, and stepper motor are mounted on a platform together with a CCD camera for viewing the projected stripe on the collar. The camera is angled in relation to the platform and laser components to achieve a field of view (FOV) adequate to

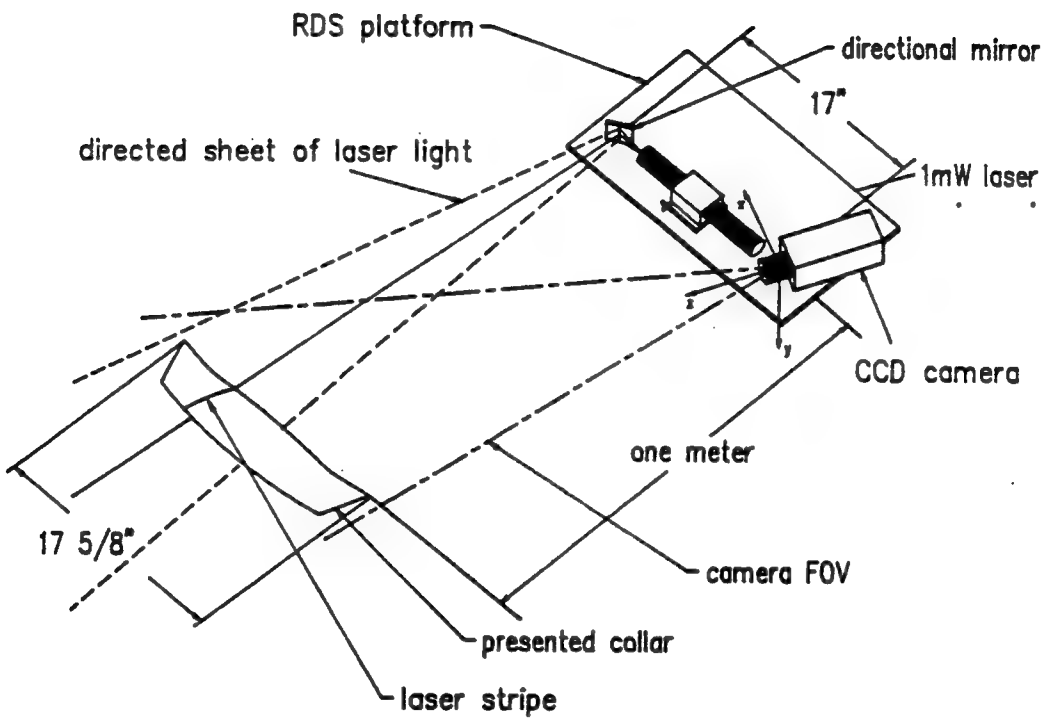


Figure 3.1. The Range Data Scanner with collar.

capture an image of the full collar as presented by the robot for loading. Figure 3.1 shows a collar positioned for range sensing.

Four key parameters are important in designing the geometrical configuration of the RDS and its components:

(1) the **base** of the triangle designated by the distance between the point of reflected laser light M and the camera focal point FP, shown in Figure 3.2,

(2) the **direction** of the field of view (FOV) as determined by the mounted camera angle,

(3) the **resolution** of the mirror rotation angle, and

(4) the **pixel resolution** of the camera image.

These design parameters each contribute to the capability and accuracy of the range finder. It is advantageous to locate the RDS in close proximity to the collar for good visual resolution of the presented collar allowing more pixels per viewed area. It is equally important to keep the full collar in camera view to adequately image the collar points. Additionally, the collar must be located symmetrically about a line extended from the mirror center to enable the scanned pattern on the collar to be symmetrical and easier to model.

Though several range finder configurations are possible that satisfy these criteria, the RDS was designed with a 23.5 degree camera angle to give a full view of the collar while maintaining the collar symmetry about a line extended from the mirror center and perpendicular to the RDS platform as shown in Figure 3.2. The perpendicularity constraint is necessary for aligning the RDS instrument in the AAW workspace. The RDS instrument was mounted on a platform designed

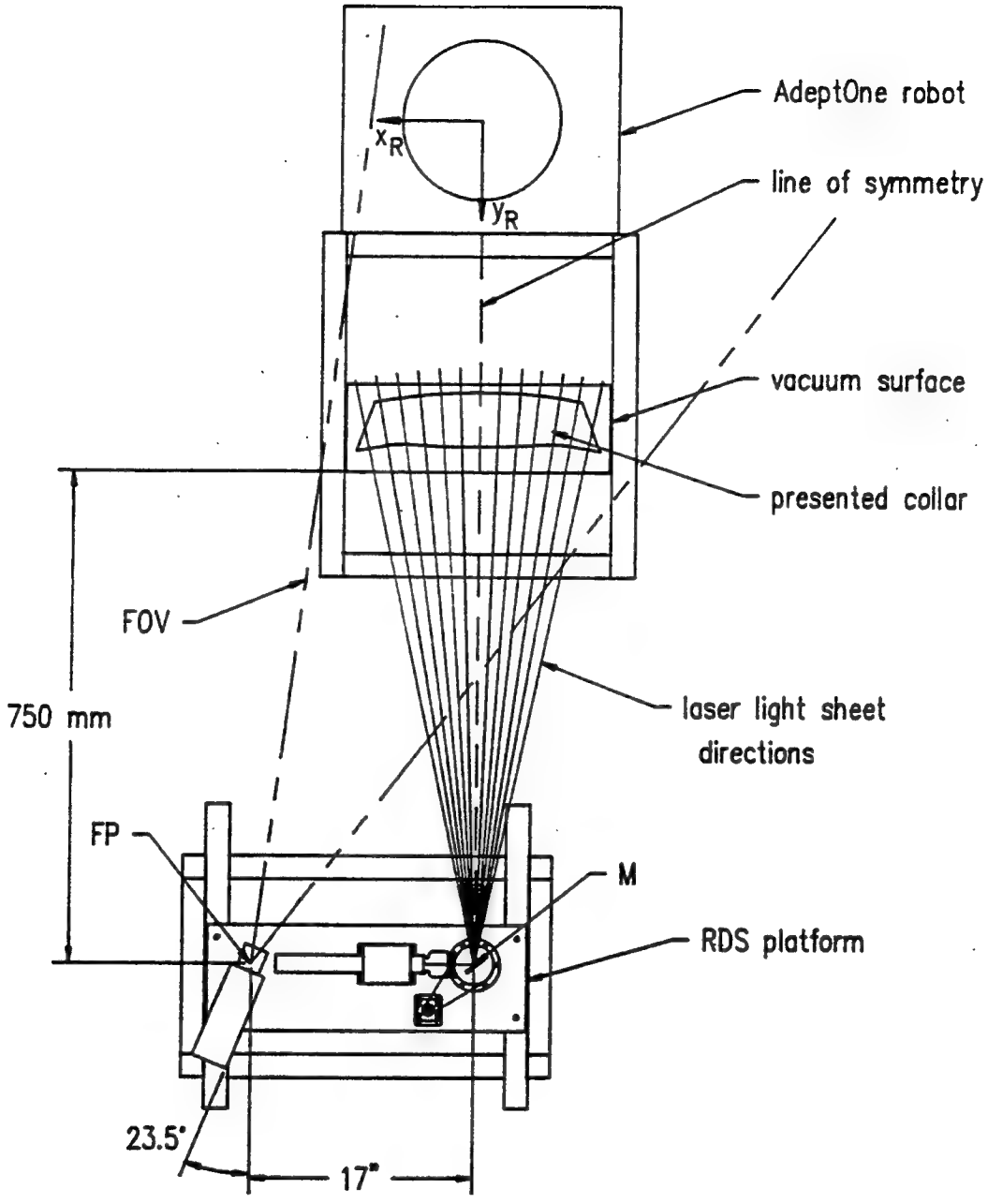


Figure 3.2. The RDS in the AAW workspace.

for adjustment in height, pitch angle, and distance from the observed collar.

Figure 3.3 depicts the component configuration and basic geometry for the RDS design. Dimension d, the base of the triangle, is the distance between the camera focal point FP and the point of laser sheet reflection on rotational mirror M. Angle θ denotes a base angle of the triangle, measured between the reflected laser sheet direction and the triangle base. This angle is controlled by the stepper motor geared to the rotational mirror. Angle γ is a fixed angle between the camera z axis (depth axis z_{RDS}) and the triangle base. Angle Φ is determined by a function of the camera focal length f and the pixel $p(x,y)_i$ position corresponding to the imaged stripe at position $P(x,y,z)_{RDS}$ in the workspace. Angle Φ is added to fixed angle γ for the other triangle base angle. This satisfies the three required parameters for triangulation: the base length and two base angles. The fixed angle γ and base length d were measured from the RDS platform layout and camera geometry with the aid of a software drafting package. The camera geometry, involving focal length f and focal point FP, was determined experimentally by modeling the CCD camera with a pinhole camera. This work is given in Appendix B.

The object point $P(x,y,z)_{RDS}$, as shown in Figure 3.3, is a point in the 3-D view space of the camera and can be considered a point on the laser stripe projected on the collar. Point $P(x,y,z)_{RDS}$ is measured in 3-D space by the RDS with

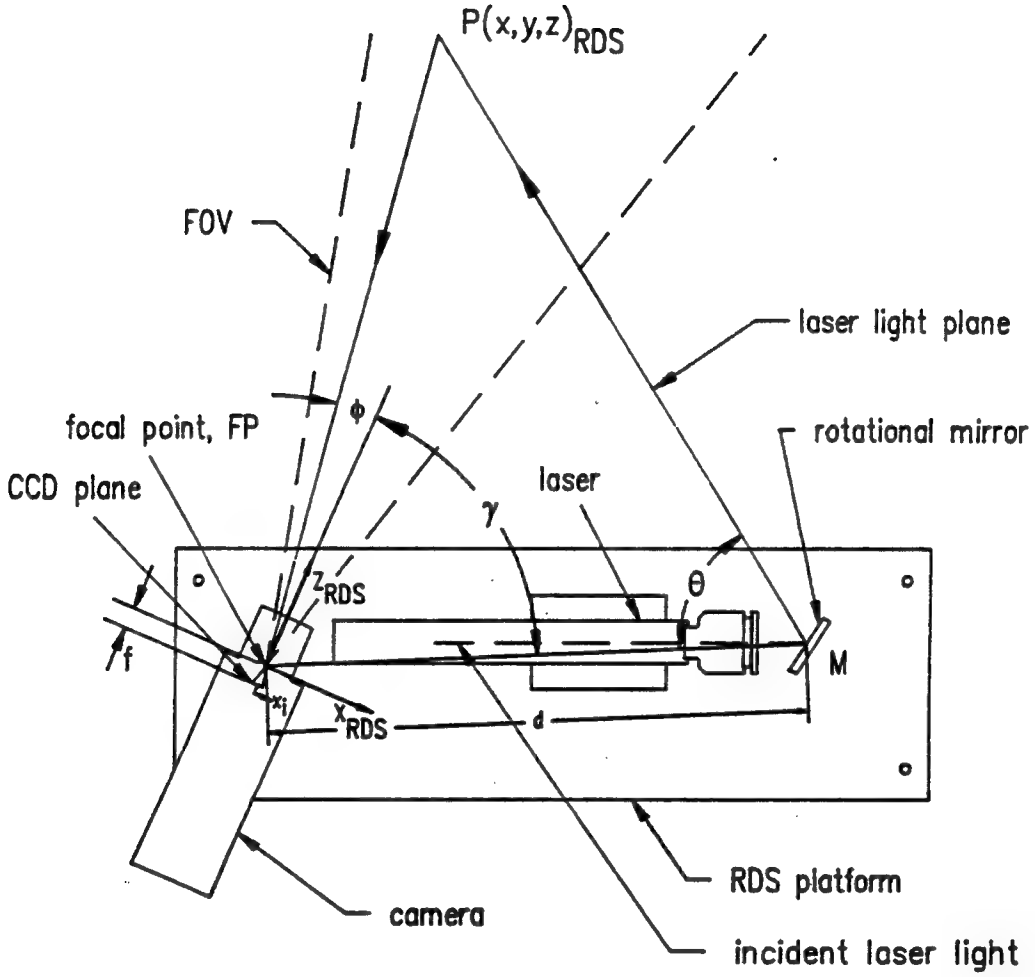


Figure 3.3. RDS component configuration and basic geometry.

coordinates defined in the RDS coordinate frame. The RDS frame origin at point FP is based at the camera focal point, with the z_{RDS} axis directed toward the object along the camera axial axis. This frame is shown in Figure 3.4, along with the RDS geometry used for triangulation. Pixel $p(x,y)_i$ on the CCD image plane corresponds to object point $P(x,y,z)_{RDS}$ and has x_i and y_i components measured in millimeters from the center of the image frame. The angle Φ is computed using the pin-hole camera model geometry using

$$\Phi = \arctan2(x_i/f). \tag{3.1}$$

The mirror angle, which is controlled by the stepper motor driver in half-step mode, is rotated in increments of 0.225 degrees (measured clockwise in Figure 3.4). The mirror datum of 0.0 degrees corresponds to the incident laser sheet direction. The angle of reflected light from a mirror equals the angle of incident light; therefore, the reflected laser sheet is controlled in increments of 0.45 degrees, or double the mirror rotation angle. An offset angle of 2.47 degrees between the triangle base and the line of incident laser light is necessary to realize the complete base angle θ since the focal point FP is not in line with the incident laser sheet plane.

The RDS triangulation for detecting the 3-D coordinates of the object point $P(x,y,z)_{RDS}$, as depicted in Figure 3.4, involves a set of basic trigonometric relations. These equations are

$$d = d_1 + d_2, \tag{3.2}$$

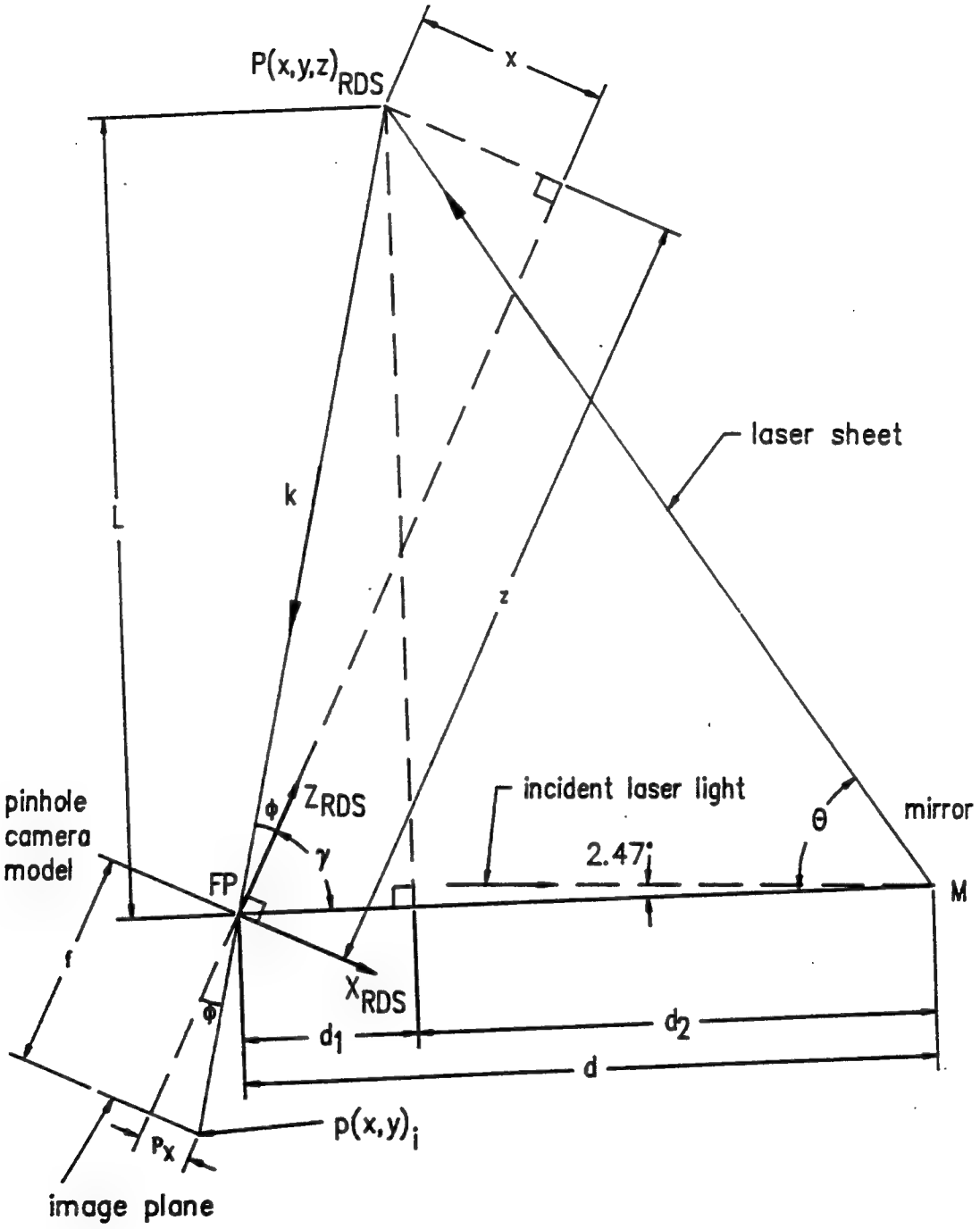


Figure 3.4. RDS geometry for triangulation.

$$\tan(\gamma+\Phi) = L/d_1, \text{ and} \tag{3.3}$$

$$\tan\theta = L/d_2. \tag{3.4}$$

which combine to yield the equality

$$L = d_1 \cdot \tan(\gamma+\Phi) = d_2 \cdot \tan\theta. \tag{3.5}$$

Rearranging equation (3.5) and including equation (3.2) gives

$$d_1 = d \cdot \tan\theta / \{\tan(\gamma+\Phi) + \tan\theta\}. \tag{3.6}$$

Since triangle side k is expressed as

$$k = d_1/\cos(\gamma+\Phi), \tag{3.7}$$

distance z_{RDS}, which is the depth of point $P(x,y,z)_{RDS}$ along the camera z axis with reference to the RDS coordinate frame, is represented as

$$z_{RDS} = k \cdot \cos\Phi. \tag{3.8}$$

By combining equations (3.6), (3.7), and (3.8),

$$z_{RDS} = d(\tan\theta)(\cos\Phi)/\cos(\gamma+\Phi)\{\tan(\gamma+\Phi) + \tan\theta\}. \tag{3.9}$$

The x_{RDS} and y_{RDS} coordinates of point $P(x,y,z)_{RDS}$ are determined using

$$x_{RDS} = z_{RDS} \cdot (x_i/f), \text{ and} \tag{3.10}$$

$$y_{RDS} = z_{RDS} \cdot (y_i/f). \tag{3.11}$$

This completes the range data transformation from image coordinates x_i, y_i, and mirror angle θ to RDS coordinates x_{RDS}, y_{RDS}, and z_{RDS}. Thus, the RDS generated range data, i.e. $P(x,y,z)_{RDS}$, are defined with respect to the RDS coordinate frame. The AAW workspace, however, is defined by the Adept-One robot coordinate frame. To evaluate robot trajectories for loading the collar on the vacuum surface, the geometry of the collar model must be defined in robot coordinates. Since the collar is sensed with the RDS, the resulting range data

must be transformed to workspace coordinates with a coordinate transformation determined by calibrating the RDS with robot coordinates.

RDS Integration

Figure 3.5 shows the RDS positioned in the workstation. The distance separating the camera and collar is in the range of one meter to maintain both the symmetrical striping capability and a full collar view. The one meter viewing range is established by the RDS position and the collar presentation position which is located overhead the target position on the pressing device. This position provides for a short robot trajectory to load the collar onto the vacuum surface. With the RDS positioned approximately one meter from the presented collar position, the RDS platform is located outside the robot workspace.

To calibrate the RDS coordinate system with respect to the robot reference frame, it is necessary for the two systems to identify common points in the 3-D workspace. An intermediate coordinate frame, which can be identified by both systems, is used to transform RDS range data to robot coordinates. Figure 3.5 shows an end-effector coordinate frame "EE_frame", which is an intermediate frame between the RDS and the robot coordinate system labeled RDS and R respectively. Since the possibilities for intermediate frames are infinite, EE_frame is selected for measurement by the calibration pointer which is shown in Figure 3.6.

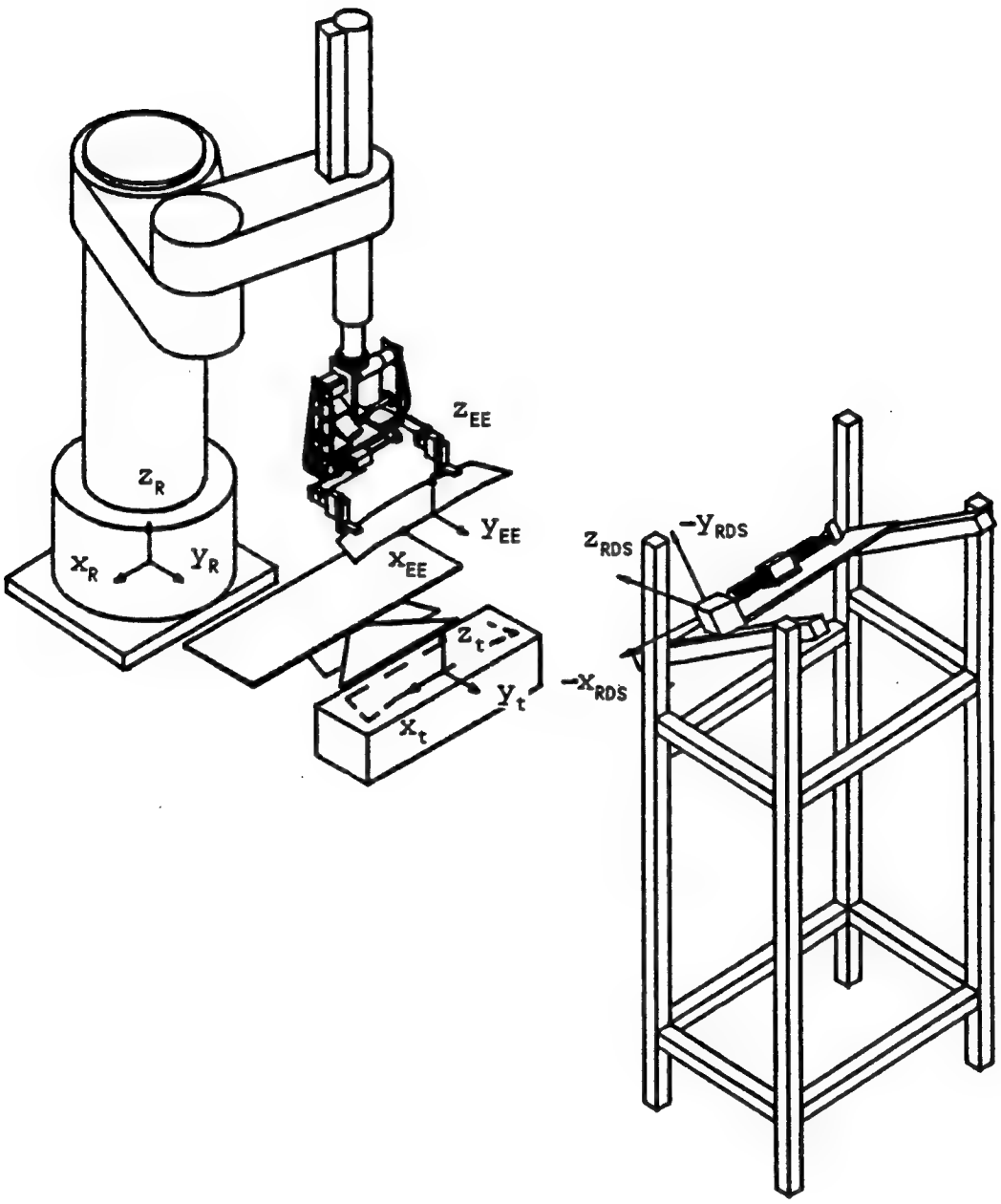


Figure 3.5. RDS, AdeptOne robot, and workspace.

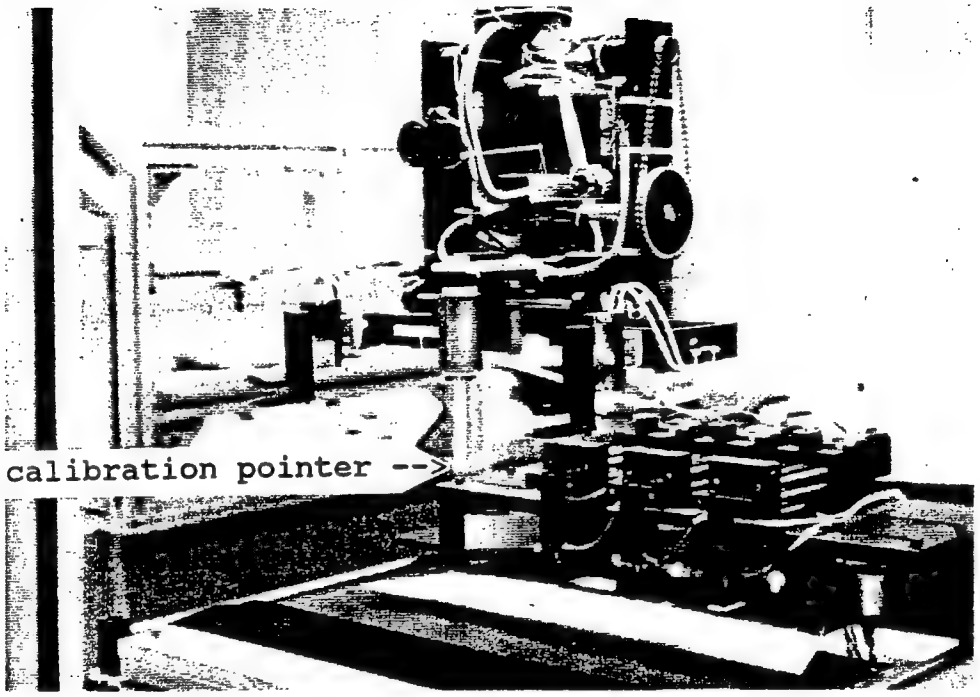


Figure 3.6 The end-effector with calibration pointer.

The calibration pointer, which is detachable, has been developed for the end-effector to point to known locations in the robot workspace. The pointer has a pointed end whose position is known through the robot forward kinematics [17] enabling point identification in the robot coordinate frame. RDS calibration is accomplished by guiding the pointer through a discrete set of points along the axes of an imaginary intermediate frame EE_frame defined in robot coordinates. The corresponding RDS range data for each point is determined by triangulation, and is used to evaluate a homogeneous transformation matrix for converting range data from RDS to robot coordinates. Appendix C details this calibration process and the matrix algebra involved. The resulting relation (C.6) from Appendix C is given by

$$V_R = {}^{R}T_{RDS} * V_{RDS}, \qquad (3.12)$$

where V_{RDS} is a position vector in RDS coordinates and V_R is a position vector in the robot workspace, enabling the system to develop the collar model in workstation coordinates. Both the collar and target locations are now in a common coordinate system permitting the robot trajectory to be evaluated in AAW workstation coordinates (robot coordinates).

AAW System Control

The workstation devices and their respective components are controlled by a network of two PC microcomputers and the microcomputer-based Adept MC1 robot controller as illustrated in Figure 3.7. Each computer is dedicated to specific tasks within the workstation, and communicates via an RS-232 data

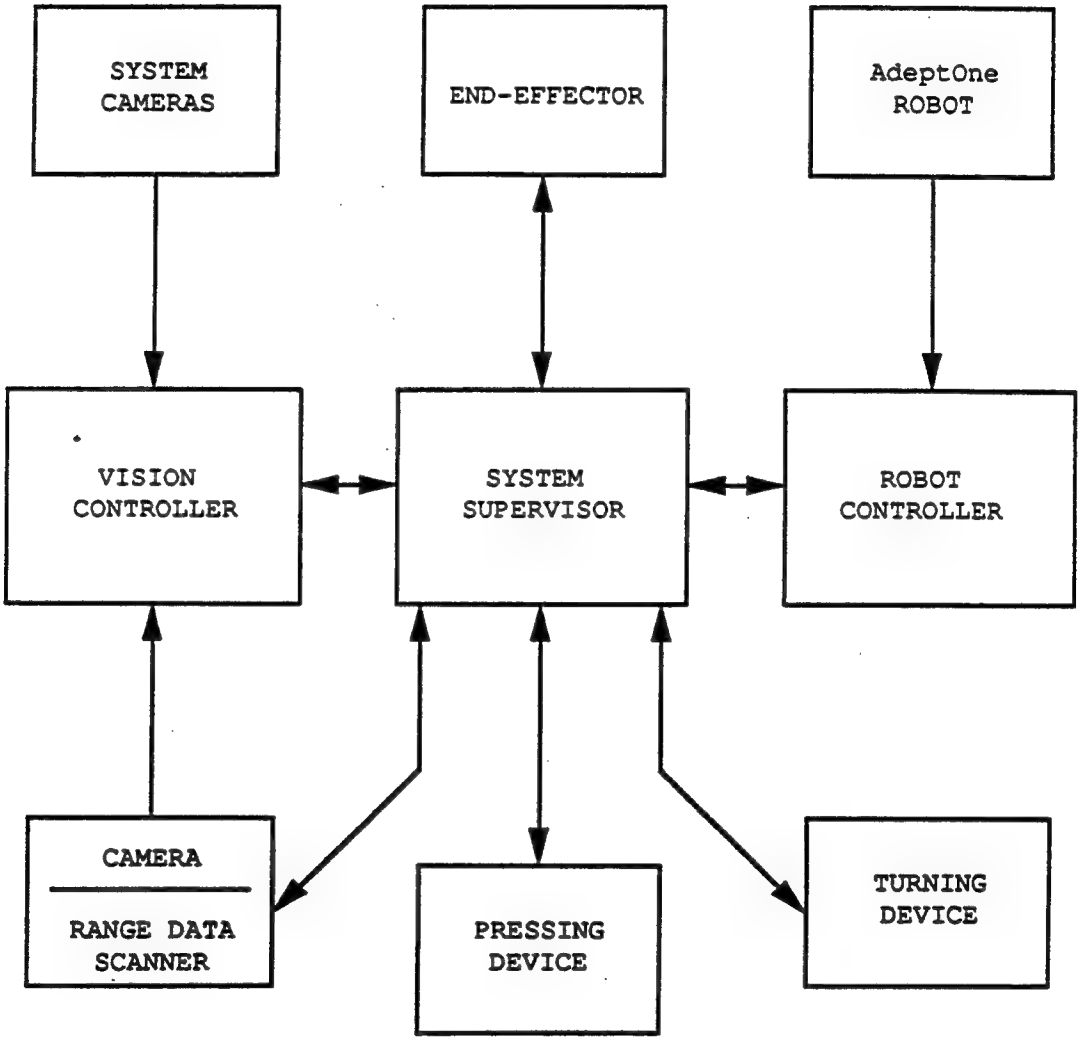


Figure 3.7. AAW configuration.

link. A PC designated the System Supervisor (SS) is responsible for directing the operation sequence of the workstation system. The SS passes data and sends synchronization signals to both the robot controller (RC) and the Vision Controller (VC), which is responsible for machine vision hardware and software activity. Additionally, the SS controls the input/output operations for proximity sensors and pneumatic actuators through a dedicated I/O board. Another board dedicated for motor control modules controls the stepper and servo motor drivers involved in the AAW. All peripheral control software and machine vision command software is accessed with programs coded in C.

The RDS is linked to both the System Supervisor and Vision Controller. The SS commands the mirror rotation with a control module linked to a chopper driver for the mirror stepper motor. An SS input port is used to monitor the output of a proximity sensor positioned to sense when the mirror shaft is at a home position. Additionally, the on/off state of the laser is controlled with an SS output signal. The camera video output signal is routed to a frame grabber board in the VC, enabling the VC to acquire images. Figure 3.8 illustrates the connection between the workstation components specifically addressed in this thesis: the RDS, creaser blades, vacuum, robot, SS, VC, and RC.

The scanning activity of laser striping is controlled by the System Supervisor, and the machine vision activity is conducted by the Vision Controller. The collar model is

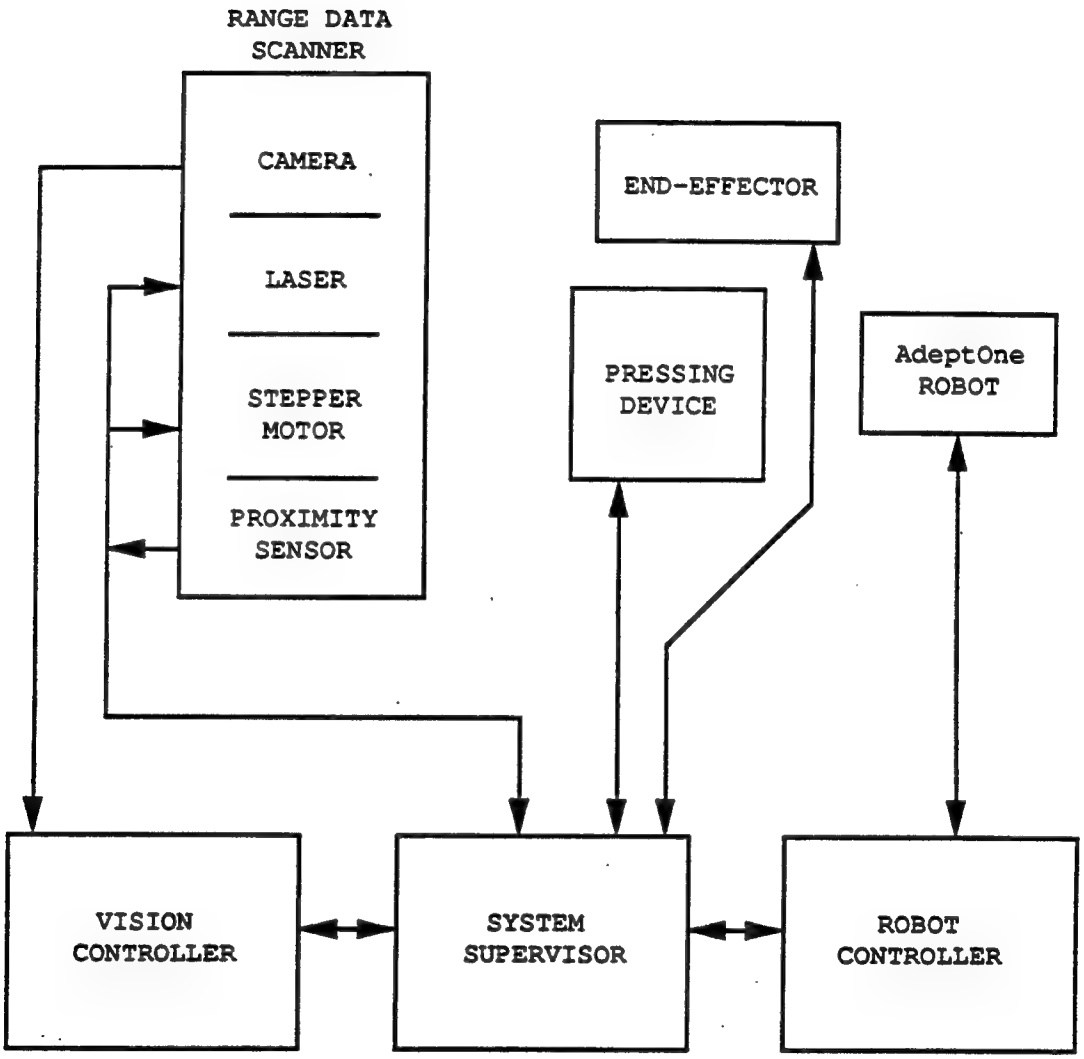


Figure 3.8 AAW components required for collar loading.

generated by the VC, an activity involving range finding and image processing computations. The VC is additionally responsible for the collar frame match discussed in Chapter II. The results of the frame match are transmitted to the RC via the SS in the form of an error vector. With the error vector, the RC develops the robot trajectories for loading the collar.

Image Processing

Image processing performs the operations to: (1) filter an acquired image for laser stripe isolation, (2) scan the image for the laser stripe location, and (3) detect the two endpoints of the stripe line. For each stripe projected on the collar, the above image processing sequence is executed by an algorithm to extract the 2-D image coordinates of the stripe endpoints. The endpoint locating module of the image processing algorithm is restricted to stripes without discontinuities, or gaps, in order to accurately evaluate the endpoints. The endpoint data, consisting of x_i and y_i image plane coordinates and the mirror angle θ, are used by the collar model algorithm to generate a collar model.

Filtering

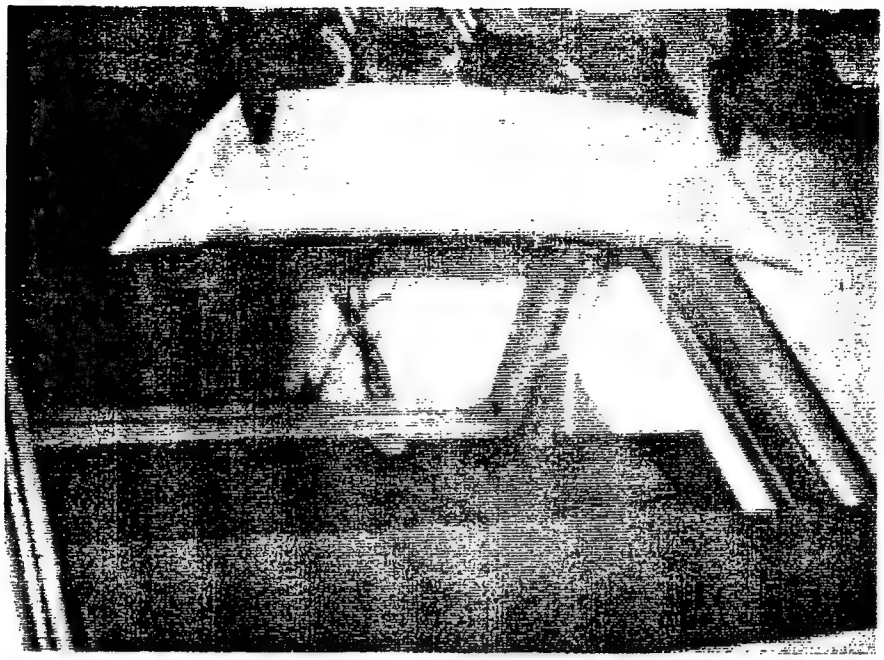
Filtering is implemented to enhance the laser stripe apart from the ambient and collar fabric backgrounds. (The color of the collar is military green; however, this procedure is designed for any color collar). The laser source has been chosen for its light intensity and non-dispersive

nature. HeNe laser light is red, with a wavelength of 632 nm. To enhance the distinct red laser stripe, a red lens filter is installed on the camera. The filter passes red light and restricts transmission of other colors, performing as an optical band-pass filter for red light. This lens filter improves the isolation of the laser stripe, but several difficulties remain in distinguishing the stripe from the background.

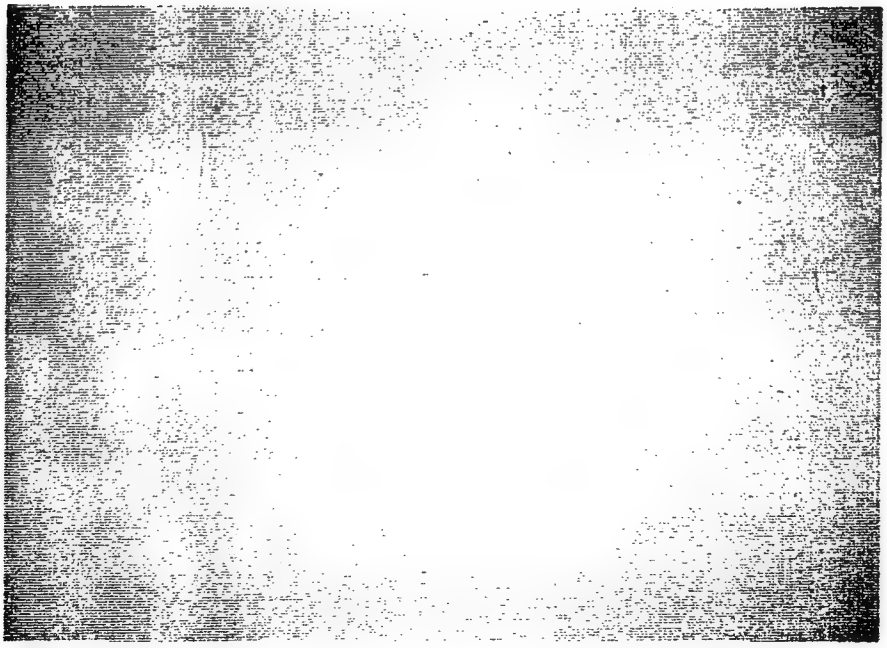
Since there is red light in the background bright enough to pass through the filter, thresholding binarizes the image to segregate the bright pixels. Pixels with intensity levels above a selected threshold level are "turned on", set to a high level of 255; and pixels with intensity levels below the threshold level are "turned off", set to 0 intensity level. Figure 3.9(a) demonstrates a laser stripe as imaged by the RDS camera through a red filter. Figure 3.9(b) shows the same stripe after thresholding. Prior to thresholding, the laser stripe is primarily constituted of bright pixels; however, due to the variability in fabric surface orientation, the possibility exists for some of the laser stripe pixels to be thresholded low. This causes discontinuities in the imaged laser stripe.

Region Growing

Several solutions are available to alleviate stripe discontinuities. Lowering the threshold level, which also thresholds less "noise", is a possibility. Another technique, region growing, is a pixel operation which takes place



(a) red filtered laser stripe



(b) thresholded stripe

Figure 3.9. Line stripe isolation.

after thresholding. Region growing involves expanding the boundary of a grouped pixel region by a given number of pixels. An imaged laser stripe can be considered a region or regions of grouped pixels. Region growing fills in the discontinuity gaps in the stripe image, providing that the gaps are not greater than twice the added boundary thickness. A pixel operator works on each pixel in the image plane by reassigning the individual pixel intensity value as a function of it and its eight neighboring pixel intensity values. The region growing operator in this application merely "turns on" the pixel if it has a neighboring pixel in the "on" state. This results in a boundary growth of one pixel layer, which will bridge a maximum gap of two pixels, including those gaps measured in a diagonal orientation.

Figure 3.10 gives a region growing example utilizing this binary (on/off) pixel operator. Figure 3.11 shows the stripe of Figure 3.9(b) after the region growing operation demonstrated in Figure 3.10. To further grow the region, the pixel operator can be applied for additional layers (multiple applications). Though region growing is valuable, it is time consuming and does more than fill gaps: it grows the endpoints further apart as well as amplify noisy pixels. In practice, a balance is required between the threshold value and the extent of region growing. A "best" performance between region growing and thresholding has been established which insures that the filtered and thresholded laser stripe

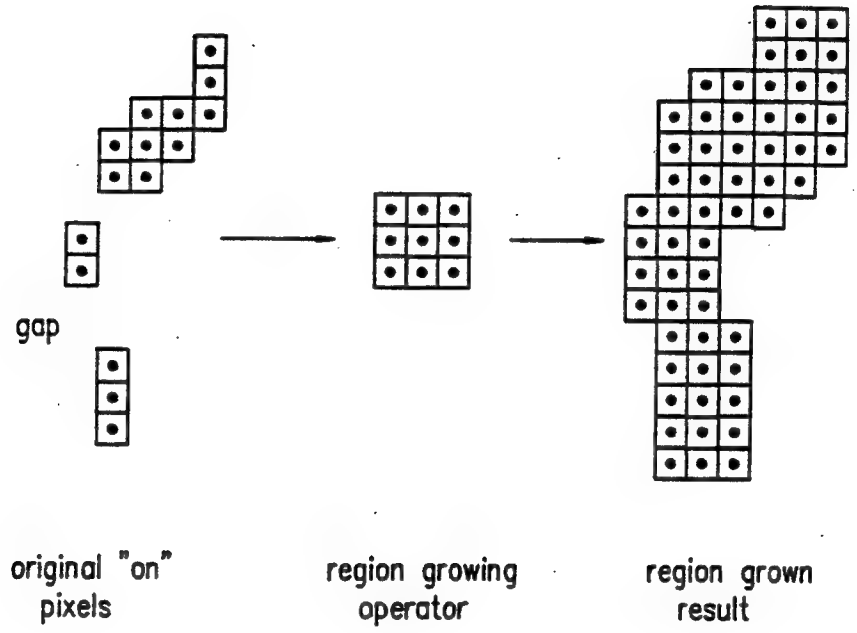


Figure 3.10 Region growing operator.

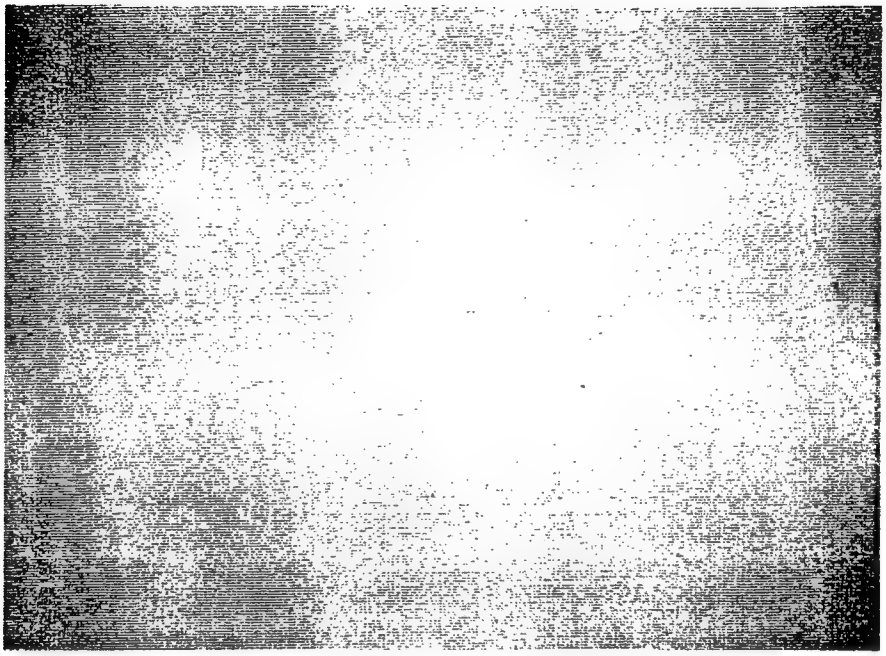


Figure 3.11 The region grown stripe of Figure 3.9(b).

can be "quickly" boundary grown to yield a continuous stripe on a blank background.

Linescan

The image processing algorithm scans the image for the stripe after image conditioning to isolate the continuous stripe. A linescan searches a row of pixels sequentially for the first "on" pixel. This pixel denotes some portion of the stripe border, indicating that the stripe has been located. A stripe is not located if the linescan fails to find an "on" pixel. In the event of failure, two subsequent linescans are attempted until the stripe is located. The subsequent scans are 15 pixels above and below the originally designated scan. The original scan is designated by a y_i-value which corresponds to a particular row of pixels in the image plane. This y_i-value is selected experimentally for each mirror angle by observing the average y_i-value for the center of the corresponding stripe projected on the collar.

Edge Detector

Once the laser stripe is located by the linescan, an edge detector, or "bug" searches for the endpoints of the stripe. The bug starts next to the located pixel, and senses its way around the boundary of the stripe until it returns to its start. The image processing algorithm tracks the travel of the bug by its x_i and y_i image frame coordinates. Since stripes are projected vertically, stripe endpoints exist at the pixels with maximum and minimum y_i-values. If several

pixels are discovered with the same y_i-value at a stripe endpoint, an average x_i-value is used for the 2-D endpoint designation. Figure 3.12 details the journey of the bug about the endpoint of a linestripe. To edgefind, the bug is programmed to travel in a clockwise manner around the region boundary; first by checking to its right; then if needed, straight ahead; then if needed, to its left to sense the boundary. If the bug is not blocked to advance (no boundary sensed), it moves by a pixel in that direction. This process is continued pixel by pixel, until the bug returns to the start.

Figure 3.13 illustrates with a flowchart the image processing algorithm executed for each stripe. Continuous stripes are tantamount to accurate stripe endpoints. Noisy pixels in the path of a linescan, on the other hand, will yield a faulty stripe detection. The key to good laser stripe data involves a trade-off between threshold value and region growing.

Collar Model Algorithm Design

The collar model, which identifies the collar geometry, discussed in Chapter II, is largely based on the locations of the two collar points. The collar point locations designate the base of the quadrilateral used for the collar model. A laser striping scheme has been developed to identify the collar points utilizing predetermined laser sheet directions (the mirror angles θ) and image processing techniques to determine the resulting laser stripe endpoints. The

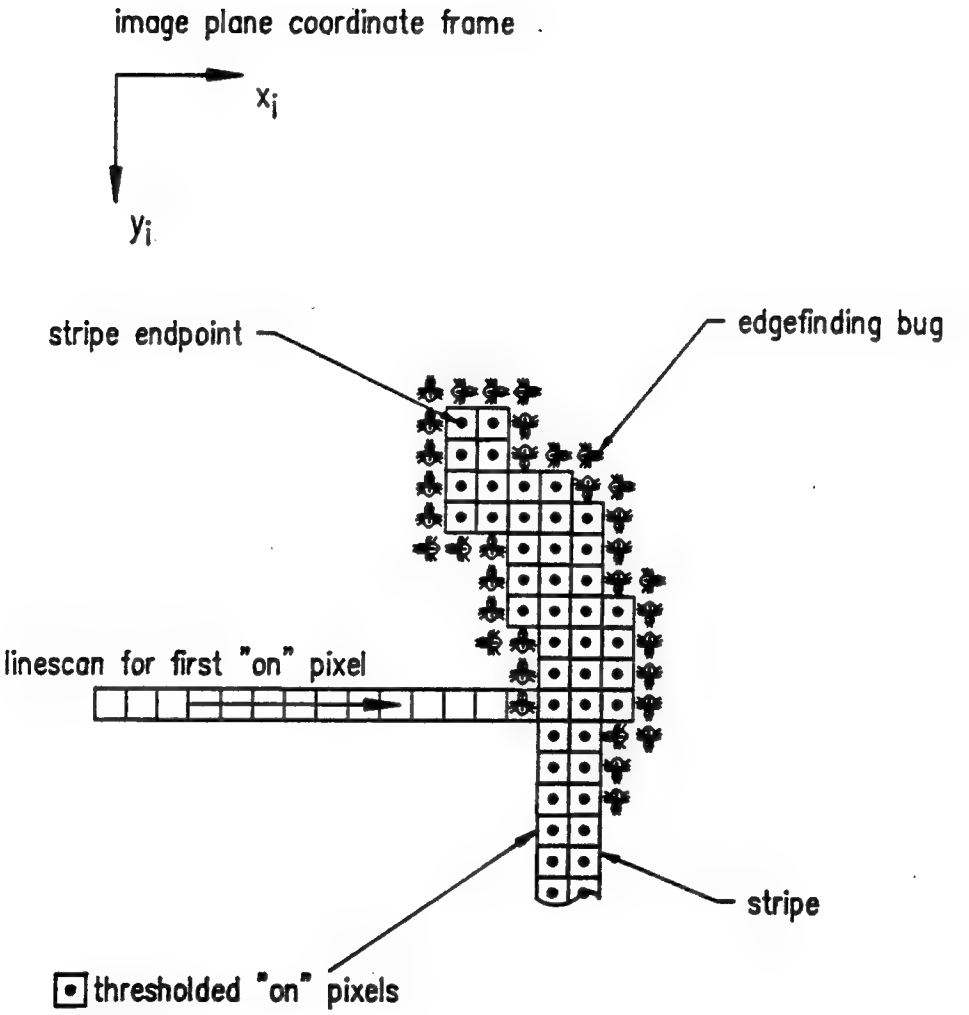


Figure 3.12. Linescan to stripe, followed by edgefind bug search for stripe endpoints.

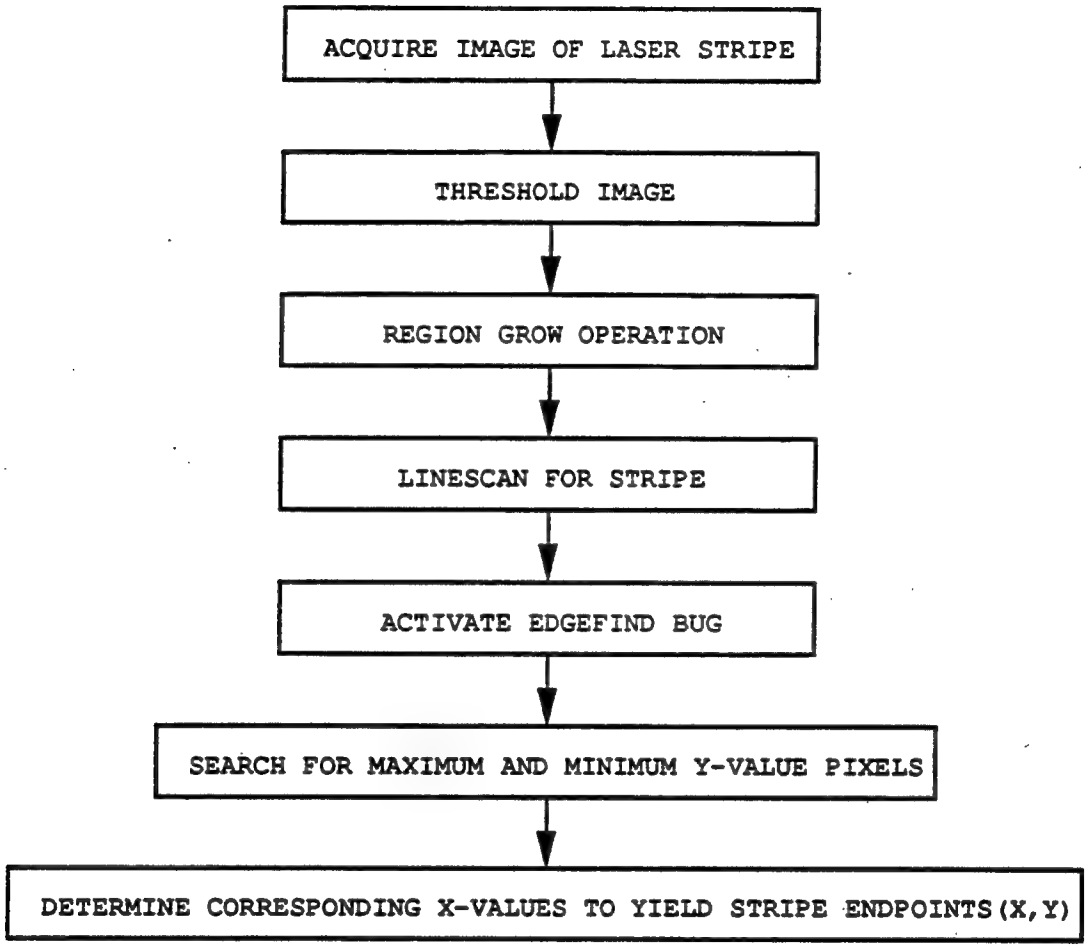


Figure 3.13. Flowchart for image processing algorithm.

endpoints are used to determine the collar points, and in turn, the collar points designate the collar ID_frame. Figure 3.14 presents a flowchart which summarizes the collar model algorithm.

Figure 3.15 depicts the general stripe placement and order of striping for a typical presented collar. The stripe locations per collar depend on the position of the collar in the workspace, since the light sheet directions remain consistent due to predetermined mirror angles. It has been confirmed that the end-effector can retrieve the collar from the turning machine with an adequate degree of consistency to ensure that the stripes will all be present on the collar, as each stripe is necessary in locating the collar points.

A collar point is defined with three laser stripes as shown in Figure 3.15. Each stripe, which is viewed individually on the camera image plane, is defined by the mirror angle θ used to generate the stripe, and its endpoint locations which are defined by 2-D coordinates relative to the image plane. The maximum y_i-value endpoints for the three stripes are used to approximate the local contour of the collar border nearest the base of the quadrilateral; and the minimum y_i-value endpoints are used to approximate the contour of the adjoining collar border. These borders, which can be approximated as straight lines, intersect at the collar point. A least squares linear regression for the maximum endpoints yields an equation approximating the local border nearest the baseline, and a second regression for the minimum

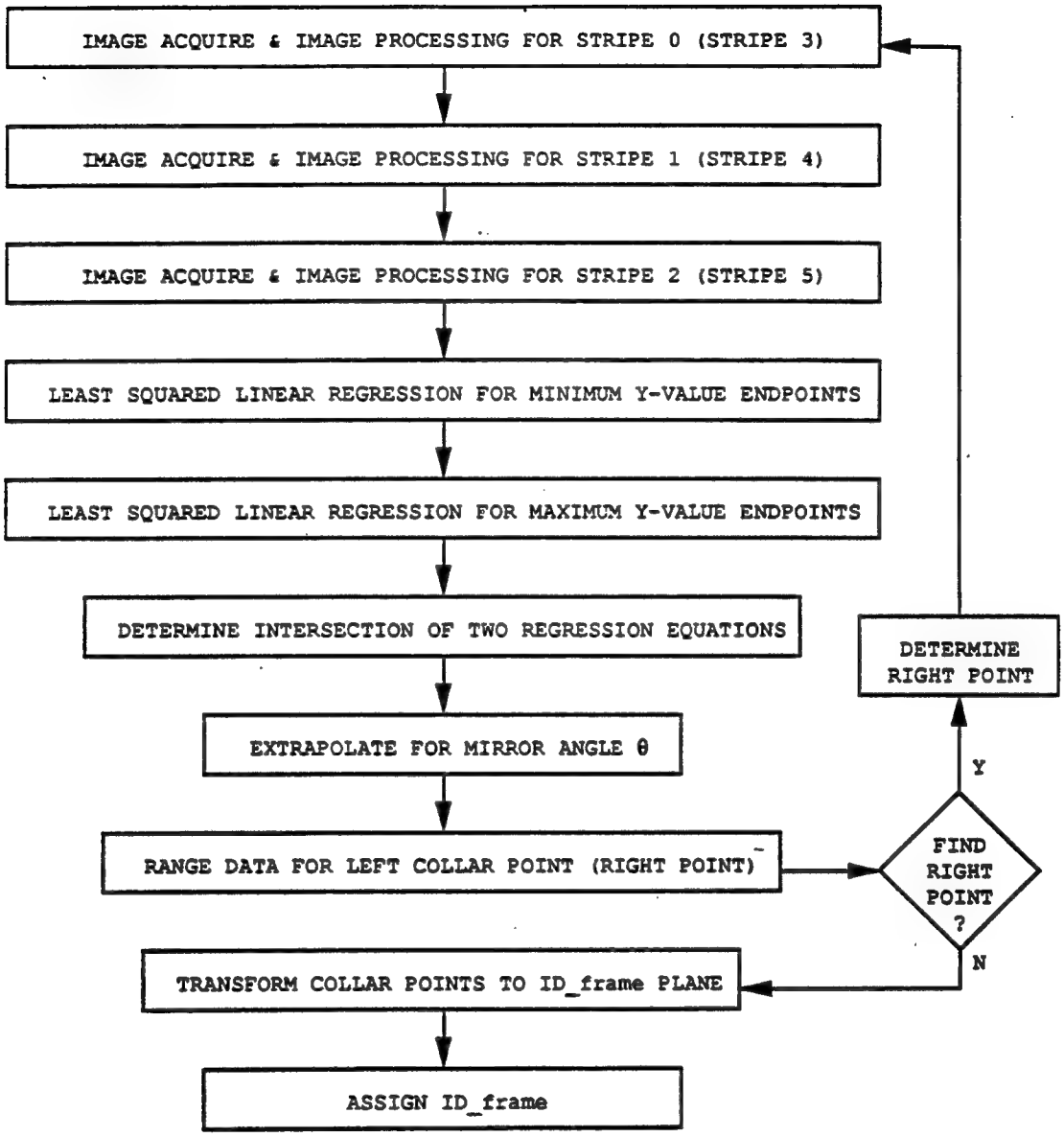


Figure 3.14. Flowchart for collar model algorithm.

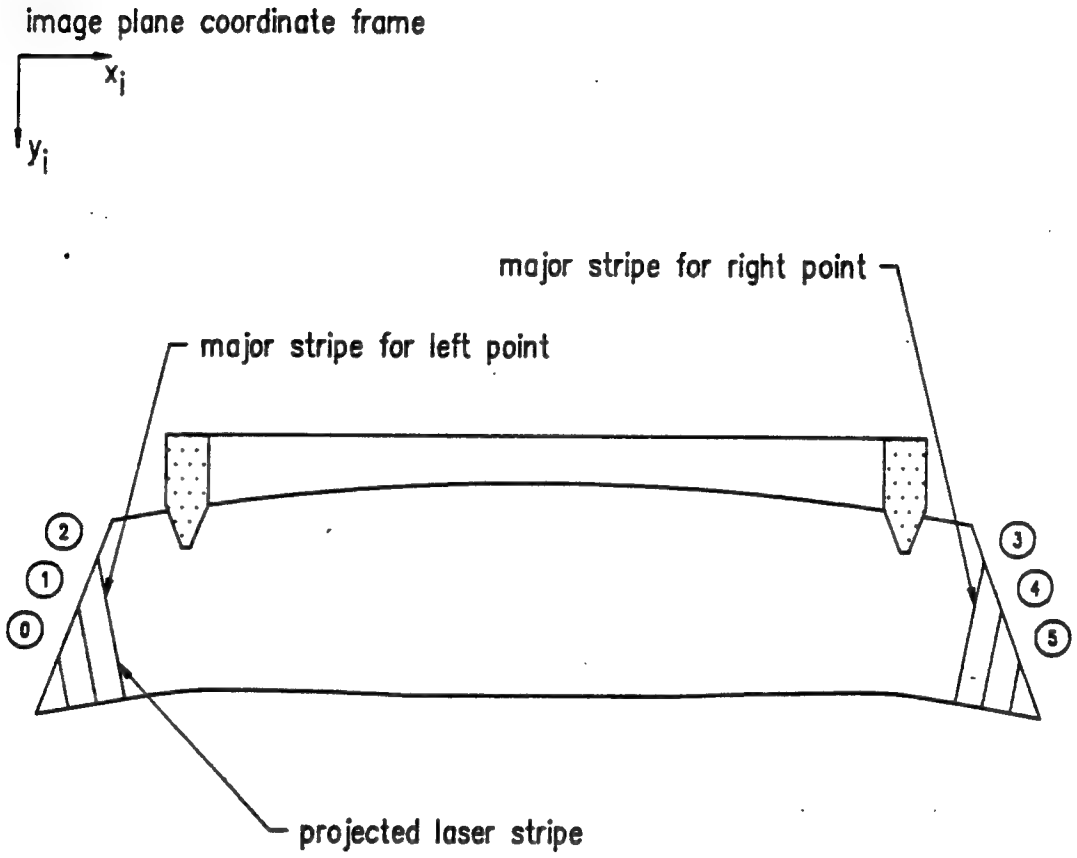


Figure 3.15. Stripe placement for a presented collar.

endpoints yields an equation approximating the adjoining border. By solving the simultaneous line equations, an intersection is determined which approximates the collar point. Figure 3.16 shows the regressions and resulting intersection for the left collar point. This intersection is a 2-D estimate of the collar point, but a 3-D location is required. The required third dimension is a function of the mirror angle, since the RDS provides range data for points defined by x_i and y_i image frame coordinates and a corresponding laser sheet projection angle θ as given by equations (3.9), (3.10), and (3.11).

Since a stripe has not been displayed at the line intersection, and indeed, very likely cannot be projected in this precise direction due to the resolution of the stepped mirror rotation, angle θ is estimated. The mirror resolution corresponds to approximately a 1/4 inch minimum stripe spacing for a typical collar one meter from the RDS. An estimate for angle θ involves extrapolating data from the three existing stripes to derive the theoretical fractional rotation required for the mirror to project the laser sheet through the 2-D intersection, designating the actual collar point. Figure 3.17 demonstrates the method for extrapolating the required stripe location and hence the angle θ to complete the collar point identification.

Two stepper motor half-steps geared to the mirror rotation are used for spacing the stripes; these half-steps which cause a change in angle θ are correlated to the measured

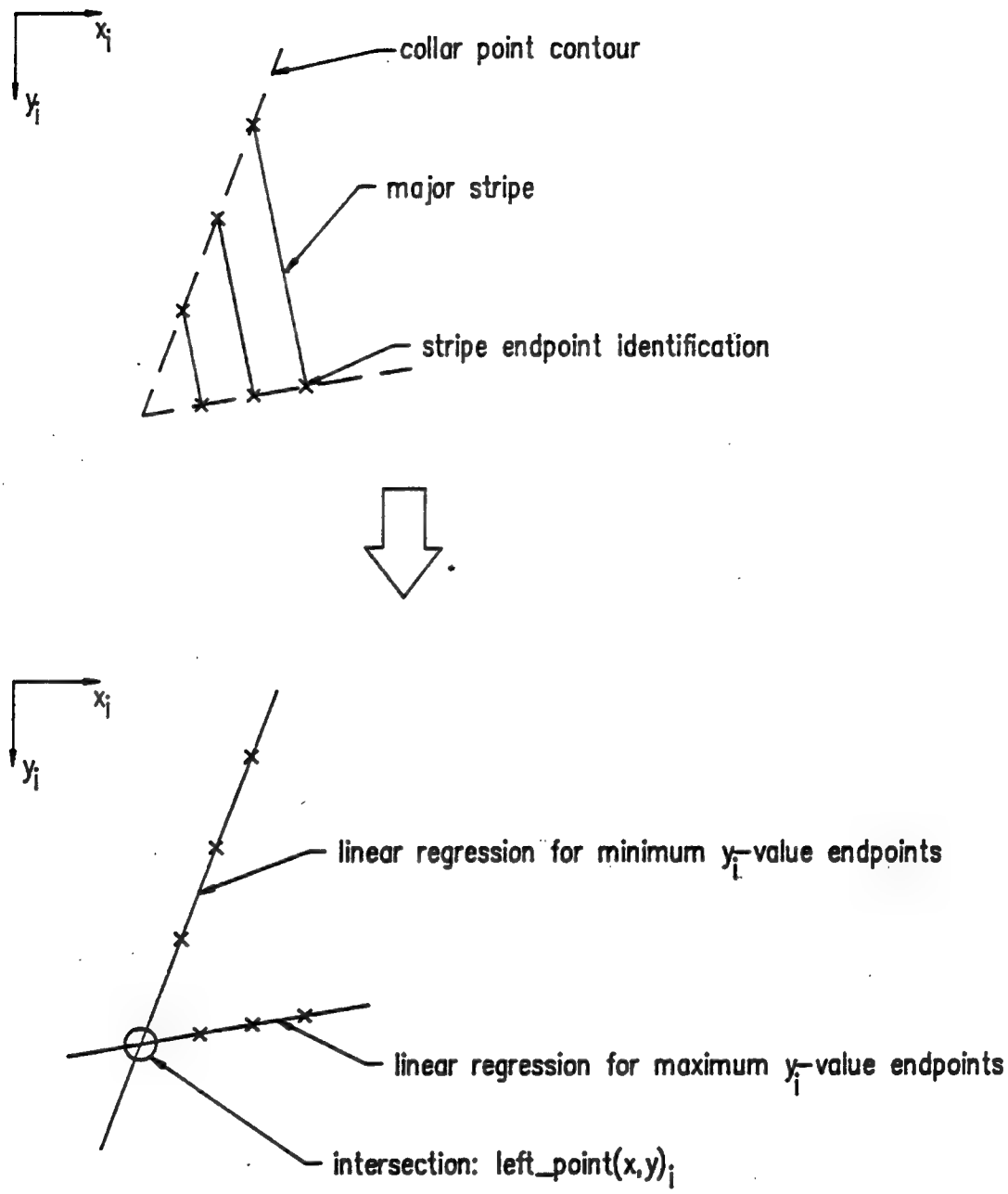


Figure 3.16. Left collar point approximation.

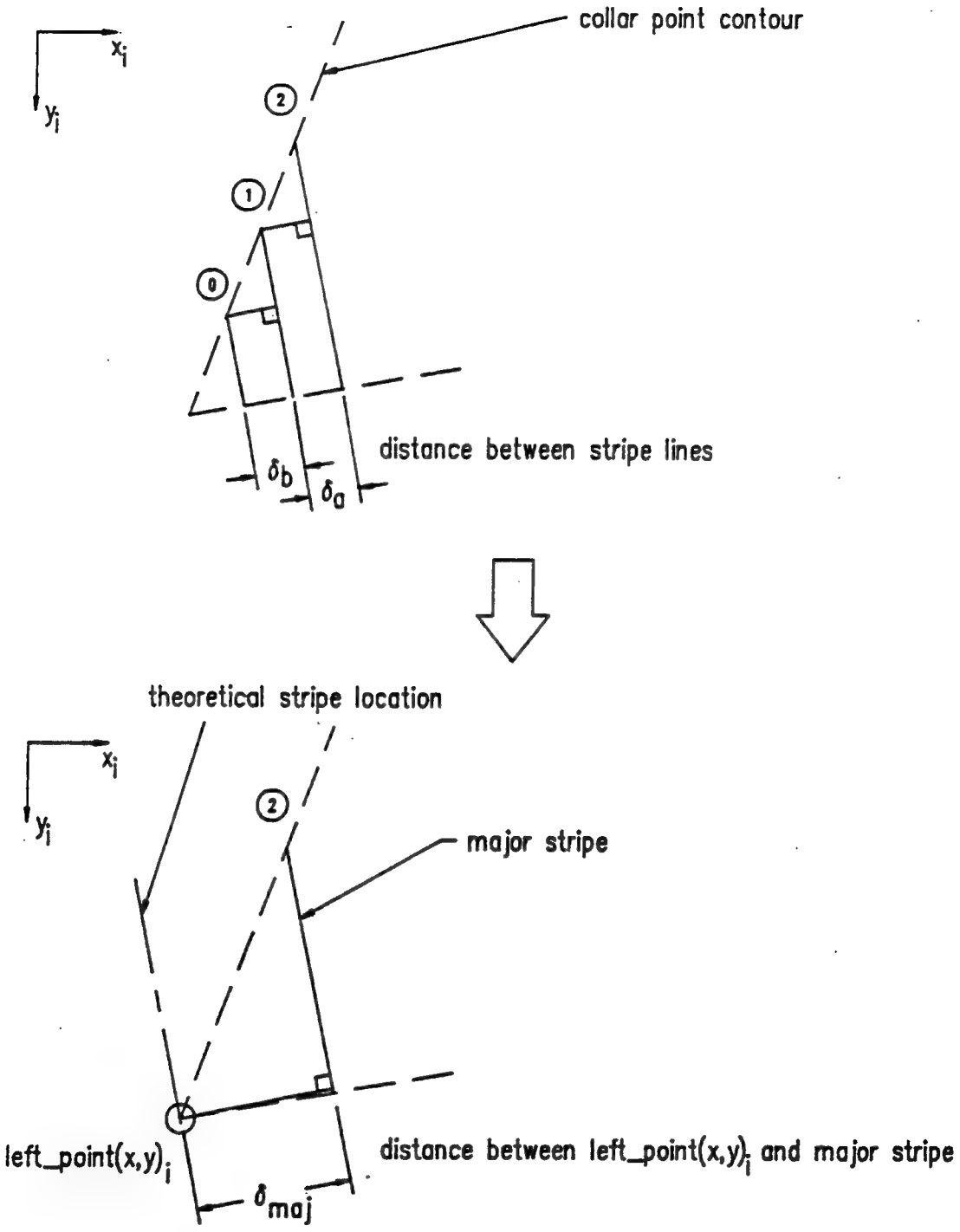


Figure 3.17. Extrapolation for corresponding stripe position.

spacing δ_a and δ_b between the imaged stripes. A ratio between half-steps and stripe separation is determined for each subsequent stripe pair and averaged to give a ratio r between change in mirror angle and change in image plane stripe location using the equation

$$r = 4 \; / \; \{\delta_a + \delta_b\} \qquad \text{(half-steps/mm)}. \qquad (3.13)$$

Finally, the perpendicular distance δ_{maj} between the line intersection $\text{left\_point}(x,y)_i$ and the major stripe is determined and correlated in fractional half-steps to yield the desired angle θ using the equation

$$\theta = \theta_{maj} - \{\delta_{maj}\; r\} \qquad \text{(half-steps)}, \qquad (3.14)$$

where θ_{maj} is the mirror angle used to project the major stripe, or stripe 2. The x_i, y_i, and θ values are triangulated with RDS geometry to estimate the 3-D location for a collar point. The same operation is employed for the right collar point with a sign change in Equation (3.14) to reflect the change in orientation of $\text{right\_point}(x,y)_i$ with respect to the right point major stripe (stripe 3).

$$\theta = \theta_{maj} + \{\delta_{maj} \cdot r\} \qquad \text{(half-steps)}. \qquad (3.15)$$

The RDS collar point coordinates are converted to robot coordinates for the collar model, which is defined in robot coordinates. A coordinate frame ID_frame is assigned to the collar model to identify the collar location in the workspace. ID_frame is determined for the collar using the quadrilateral wireframe model positioned parallel to the work surface. The parallel collar position requires the 3-D collar point positions, which are in robot coordinates, to be

rotated about the collar axis of rotation (the swing line previously shown in Figure 2.2) until the wireframe model is parallel with the work surface. Following this transformation to place all four corners of the quadrilateral in a plane parallel to the vacuum surface, the ID_frame origin is calculated equidistant from the collar points with the frame axes aligned as previously described in Figures 2.2 and 2.6. The ID_frame assignment concludes the collar model algorithm.

The Frame Match and Robot Trajectory

A flowchart for the frame match and robot trajectory operations is given in Figure 3.18. The target_frame depicted in Figure 3.19 identifies the destination on the vacuum surface corresponding to the ID_frame of a collar to be loaded. This coordinate frame is located equidistant from the creaser blade tips when in the extended position. The extended blade tip positions are measured for 3-D coordinates with the calibration pointer to obtain the invariant target_frame position.

Both target_frame and ID_frame coordinate frames are referenced to the robot frame. A comparison of the frame locations in the workspace yields a translation and rotation difference which refers to the position error ($\Delta x, \Delta y, \Delta\theta$) between the coordinate frames. The error in the z_R direction remains consistent for every collar since the parallel planes have a constant displacement between the gripper tips and vacuum surface. The linear Δx, Δy, and the rotational $\Delta\theta$

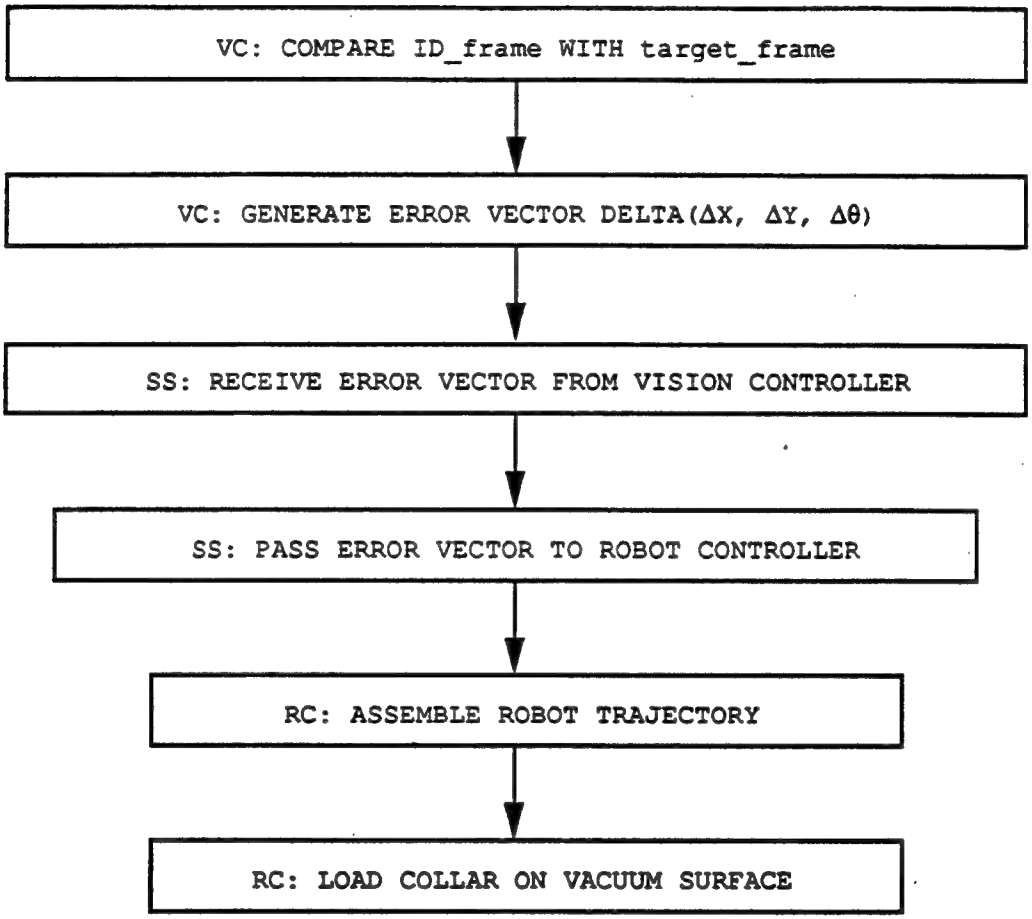


Figure 3.18. Flowchart for frame match and robot trajectory operations.

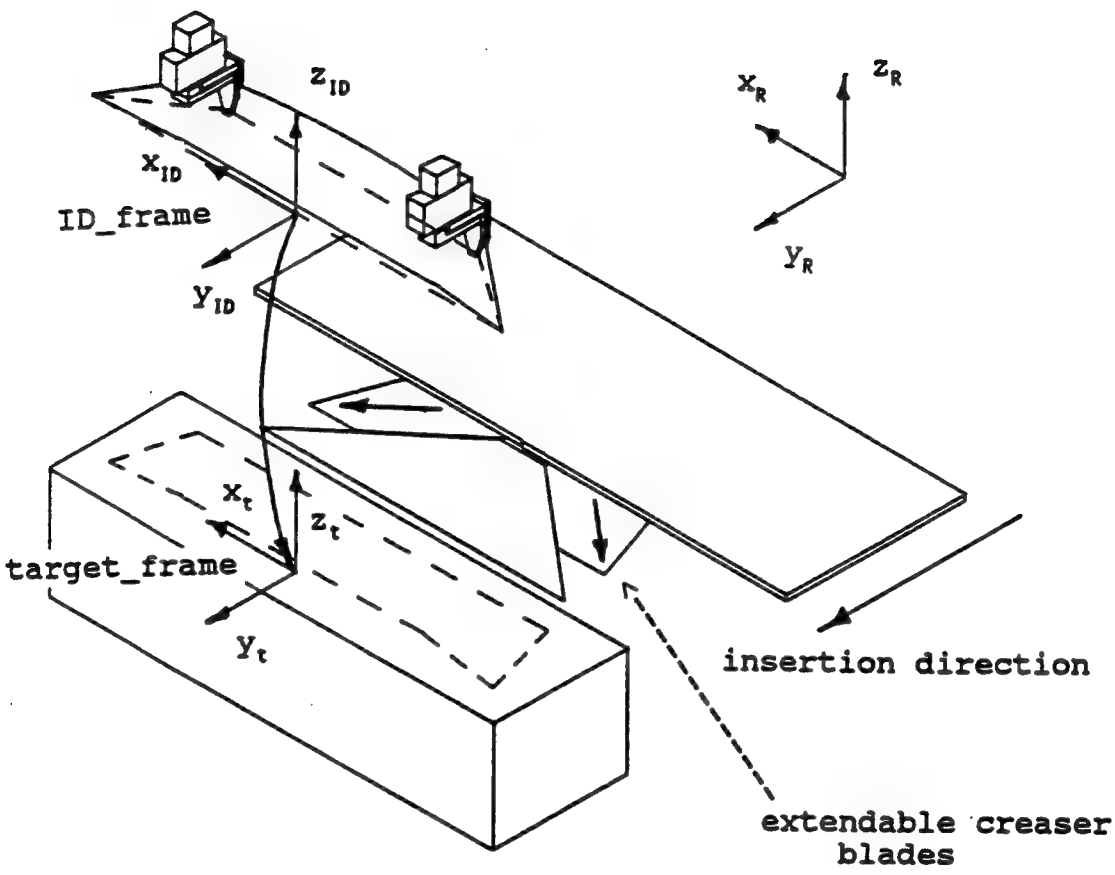


Figure 3.19. ID_frame and target_frame assignments.

displacement in the x-y robot plane are variable and dependent on the generated collar model using the equations

$$\Delta x = \text{target\_}x_R - \text{ID\_}x_R, \tag{3.16}$$

$$\Delta y = \text{target\_}y_R - \text{ID\_}y_R, \text{ and} \tag{3.17}$$

$$\Delta \theta = \text{target\_}\theta_R - \text{ID\_}\theta_R, \tag{3.18}$$

with all coordinates for the target and collar model frames referenced to the robot (R) coordinate frame.

The robot trajectory is essentially designed to "compensate the position error" or load the collar on the vacuum surface. Rather than simply command the robot to move the collar in a straightline trajectory from ID_frame to target_frame and render the wireframe model inaccurate, the collar is placed on the vacuum surface in a semi-predefined manner which is demonstrated in Figure 3.20. Figure 3.20 shows a sequence of robot and end-effector positions in the robot y-z plane along with the corresponding scaled incremental displacement vectors. These positions are labeled to correspond with the robot trajectory points produced by the robot controller source code, which is given in Appendix E.

Position loc[12] is the robot and end-effector position for the collar during the laser stripe scan by the RDS for range data. After the collar is scanned, the model generated, and the error vector passed to the robot, the robot controller develops the following progression of robot points designating the loading trajectory. The robot moves to position alt13 while the end-effector simultaneously pitches forward (as shown), performing a rotation about the collar

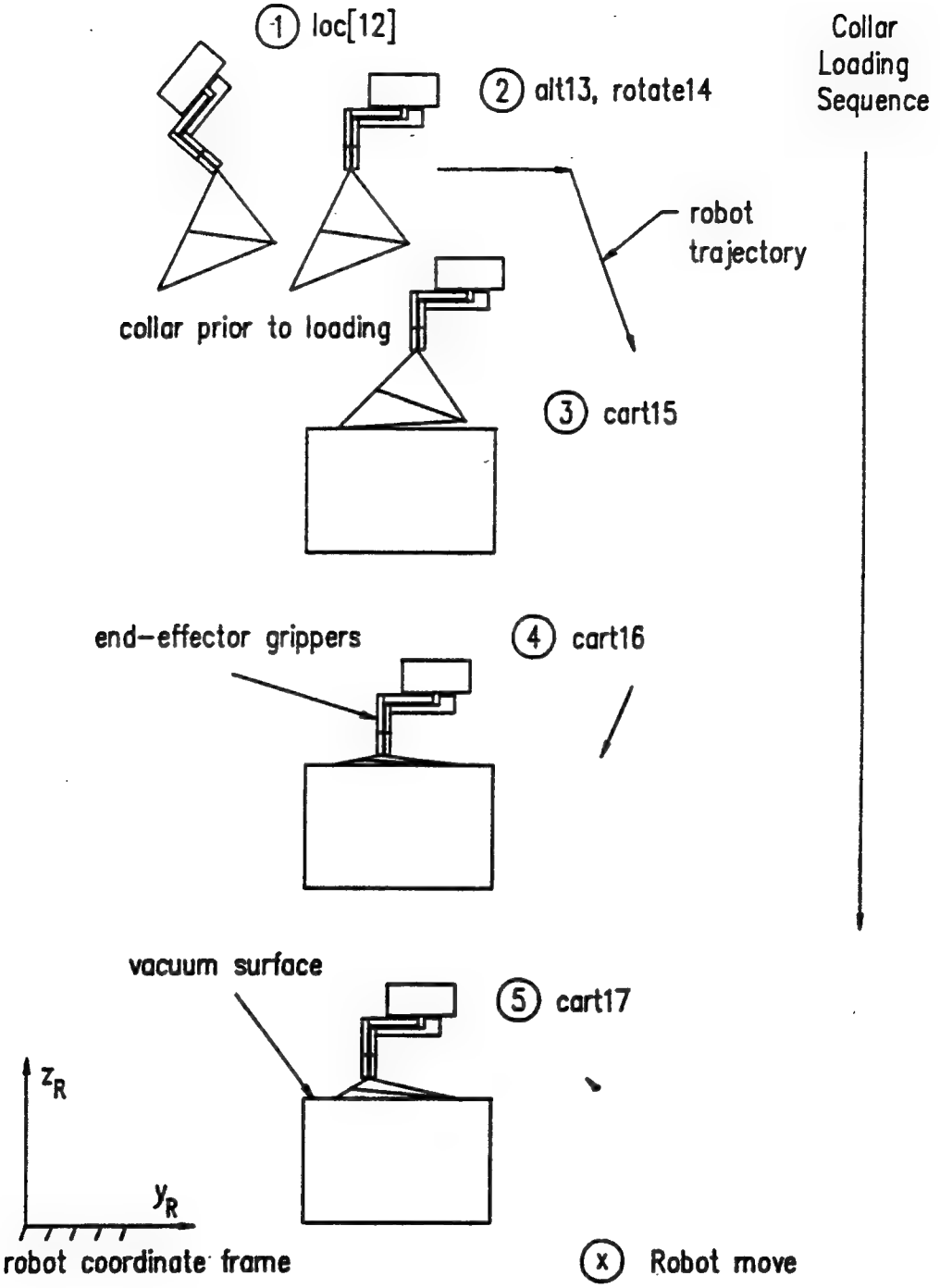


Figure 3.20. Robot trajectory to load collar (AAW sideview).

swing line. This move preserves the invariant z_R error displacement between gripper tips and vacuum surface. The robot wrist next rolls the end-effector about the ID_frame z_{ID} axis by a negative $\Delta\theta$ rotation to eliminate the rotation error.

The linear Δx and Δy errors are then compensated with a robot move to position cart15, which includes part of the z_R error displacement to place the bottom collar ply opening on the vacuum surface. At this time, the vacuum is drawn through the vacuum surface holding the open side of the bottom ply in place. Although the net Δy displacement is determined by equation (3.17) and implemented with cart15, additional predetermined y_R offset components are utilized in positions cart15, cart16, and cart17 to properly manipulate the collar to the work surface. The y_R and z_R predetermined components, which are determined experimentally, are integrated to enable the collar to be loaded without wrinkles signifying that it is loaded according to the wireframe model. The robot moves to position cart16 allowing the collar points to "lock" into position on the vacuum surface, and permitting the end-effector to flatten the collar against the vacuum surface for maximum draw on the outer opening of the collar pocket. This provides the widest possible collar opening for the creaser blades to insert. The final load position cart17 reopens the collar pocket to create a clearance between the blade surface and the collar top ply, as previously shown in Figure 2.7. In this position, the collar is loaded and ready for creaser blade insertion.

Summary of the System Operation Process

Loading a collar on the pressing device requires (1) the RDS to sense the presented collar, (2) the Vision Controller to construct a collar model and determine the collar position and orientation error, and (3) the Robot Controller to generate an appropriate robot trajectory for the robot to load the collar. The operational sequence for this task is directed by the System Supervisor. Figure 3.21 presents a flowchart to describe the complete system sequence of operation for sensing and loading a collar.

After the robot presents the collar to the pressing station, the RC signals the SS to begin laser stripe scanning on the collar. The SS, in turn, synchronizes mirror rotation commands with the image processing of the VC. Following the scanning sequence, a mathematical collar model is constructed by the VC with the stripe information and the collar model algorithm. Using the collar model, the VC assigns an ID_frame to the collar, and compares the ID_frame with the invariant target_frame to obtain a displacement error vector between the collar and target location in robot coordinates. The error vector DELTA($\Delta x, \Delta y, \Delta\theta$) is passed via an RS-232 link to the SS. The SS relays the unchanged vector to the RC, which uses the information to assemble an appropriate robot trajectory. The collar is then loaded on the vacuum surface by a synchronized RC and SS sequence.

The robot incorporates the error information in two moves, rotate14 and cart15, prior to placing the collar

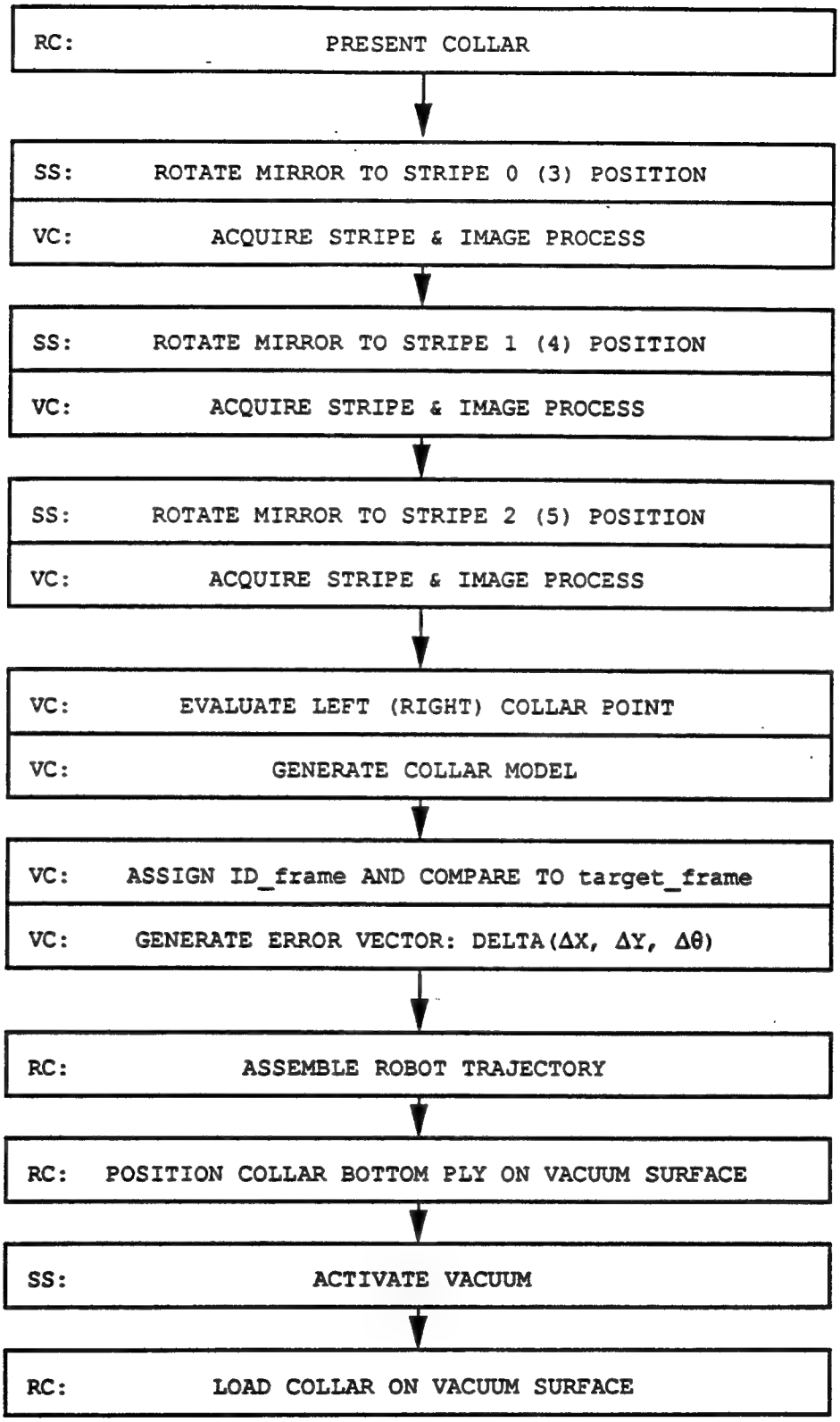


Figure 3.21 Flowchart for complete sensing and loading operation.

points on the vacuum surface. The bottom ply touches the vacuum surface with the move to position cart15 at which time the vacuum is applied by SS control. The robot completes the loading trajectory through predetermined positions cart16 and cart17 to guide the collar into its final position, preparing it for creaser blade insertion during which the vacuum draw is maintained. The robot holds the collar open until the creaser blades are inserted and extended. Prior to the creaser blade activity, however, the operation sequence is returned to SS control to begin the pressing station sequence [21] which includes the creaser blade insertion and extension.

CHAPTER IV

PERFORMANCE EVALUATION

Successful collar loading relies on the combined accuracies of the hardware and software components used for sensing the collar, generating the collar model, and determining the robot trajectory. Two system components are paramount to insuring successful collar loading: (1) the RDS for accurate range data, and (2) the collar model algorithm for accurate estimation of the collar points. Both the RDS and the collar model algorithm perform functions which identify points in the AAW workspace. The RDS measures range data points, and the collar model algorithm estimates the 3-D collar point positions. Each method for designating 3-D points features an error between measured or estimated position and the actual position. An error analysis is performed for both the RDS and collar model algorithm to analyze the accuracy of the system.

A measurement error e is composed of two components: (1) a bias error $\bar{e}$, and (2) a precision error σ_e. These two error components combine to give the measurement error e based on the equation

$$e = \bar{e} \pm \sigma_e. \tag{4.1}$$

The bias error remains constant during a given series of measurements under fixed operating conditions. Thus, in a series of repeated measurements, each measurement contains

the same amount of bias. Bias error can be estimated by comparison and is determined by calibration or with concomitant methods utilizing different physical measurement principles [18]. The bias error $\bar{e}$ is determined as the mean of the measurement errors e_i for n sample points by using equation (4.2).

$$\bar{e} = \frac{\sum_{i=1}^{n} e_i}{n}. \tag{4.2}$$

Precision error is the scatter of measurement values about the bias error. Repeatability and resolution in the measurement system are represented by the precision error. The precision error σ_e is determined by the standard deviation of the measurement errors e_i for n sample points using equation (4.3).

$$\sigma_e = sdev(e_i) = \sqrt{\frac{\sum_{i=1}^{n} (e_i - \bar{e})^2}{n-1}}. \tag{4.3}$$

Range Data Scanner Accuracy

The pinhole camera model, mirror resolution, RDS platform geometry, and image resolution all contribute to the accuracy of RDS range data measurements. Several of these parameters are more sensitive than others in terms of RDS accuracy. For example, the linear pinhole camera model does not account for lens aberration, which causes non-linearities

in the calibration of the image plane [19]. An uncertainty analysis for the RDS components including the camera model is given in Appendix B. Appendix C presents the integration of the RDS into the AAW and addresses the complexity involved in accurately calibrating the RDS to the robot coordinate system. This section evaluates the accuracy of the calibrated RDS as it performs in the workspace by investigating the error in range data measurements.

The calibration pointer was used to calibrate the RDS with the robot coordinate system, and create a transformation between RDS coordinates and robot coordinates as given by equation (3.12). A series of points in the workspace are individually measured by both the pointer/robot system and the RDS to evaluate the accuracy of the RDS as it is used in the workstation. The measurement procedure is executed by first positioning the pointer with the robot, and then using the RDS to locate the pointer tip position in 3-D space. This is demonstrated in Figure 4.1, which shows the end-effector with the laser stripe through the pointer axis. The resulting 3-D measurements for these points are compared to yield an accuracy of the RDS. The robot coordinates are transformed to RDS coordinates prior to the comparison to investigate the error results in reference to the camera coordinate frame, which is identical to the RDS coordinate frame.

The points are selected from three localized regions, representing the volumetric workspace of measurement as shown

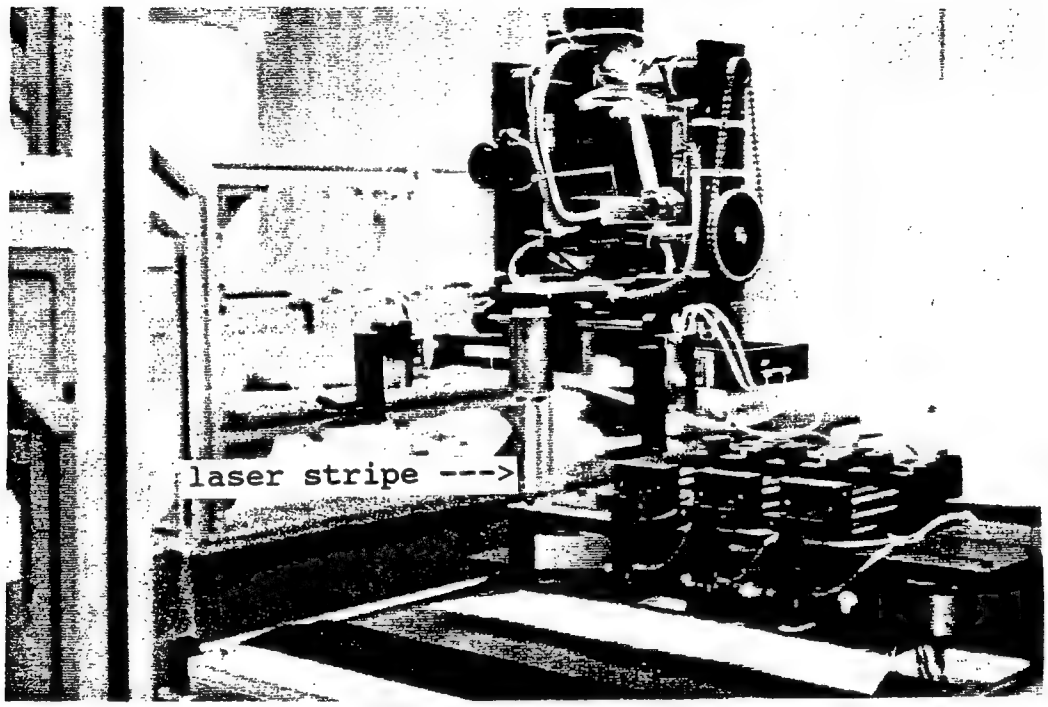


Figure 4.1. Calibration pointer with laser stripe through the pointer tip.

in Figure 4.2. Since the RDS is used to measure the collar point positions, two regions of measurement are concentrated in the left and right collar point areas. The other region is located central to the two regions closely corresponding to the intermediate coordinate frame (EE_frame) used for calibrating the RDS with the robot as shown in Figure 4.2.

Though the pointer has itself undergone a calibration procedure and is considered accurate to within ±0.5 mm due primarily to the end-effector pitch motion [17], it is considered the reference measurement to which the RDS measurements are compared. Therefore, measurement differences comparable to the error e of Equation (4.1), are evaluated per point by their error components as

$$\Delta x = x_{RDS} - x_{Robot}, \tag{4.4}$$

$$\Delta y = y_{RDS} - y_{Robot}, \text{ and} \tag{4.5}$$

$$\Delta z = z_{RDS} - z_{Robot}. \tag{4.6}$$

Each of the errors Δx, Δy, and Δz contain bias and precision error components. Figure 4.3 illustrates a geometric model describing the error components for this error analysis. An ellipsoid represents the bounded precision error of the RDS measurement offset by bias error from the measured location of the calibration pointer. A sphere is used to describe the ±0.5 precision of the calibration pointer. Table I presents the bias and precision errors calculated from the comparison between the RDS and robot measurements of common points in the three regions.

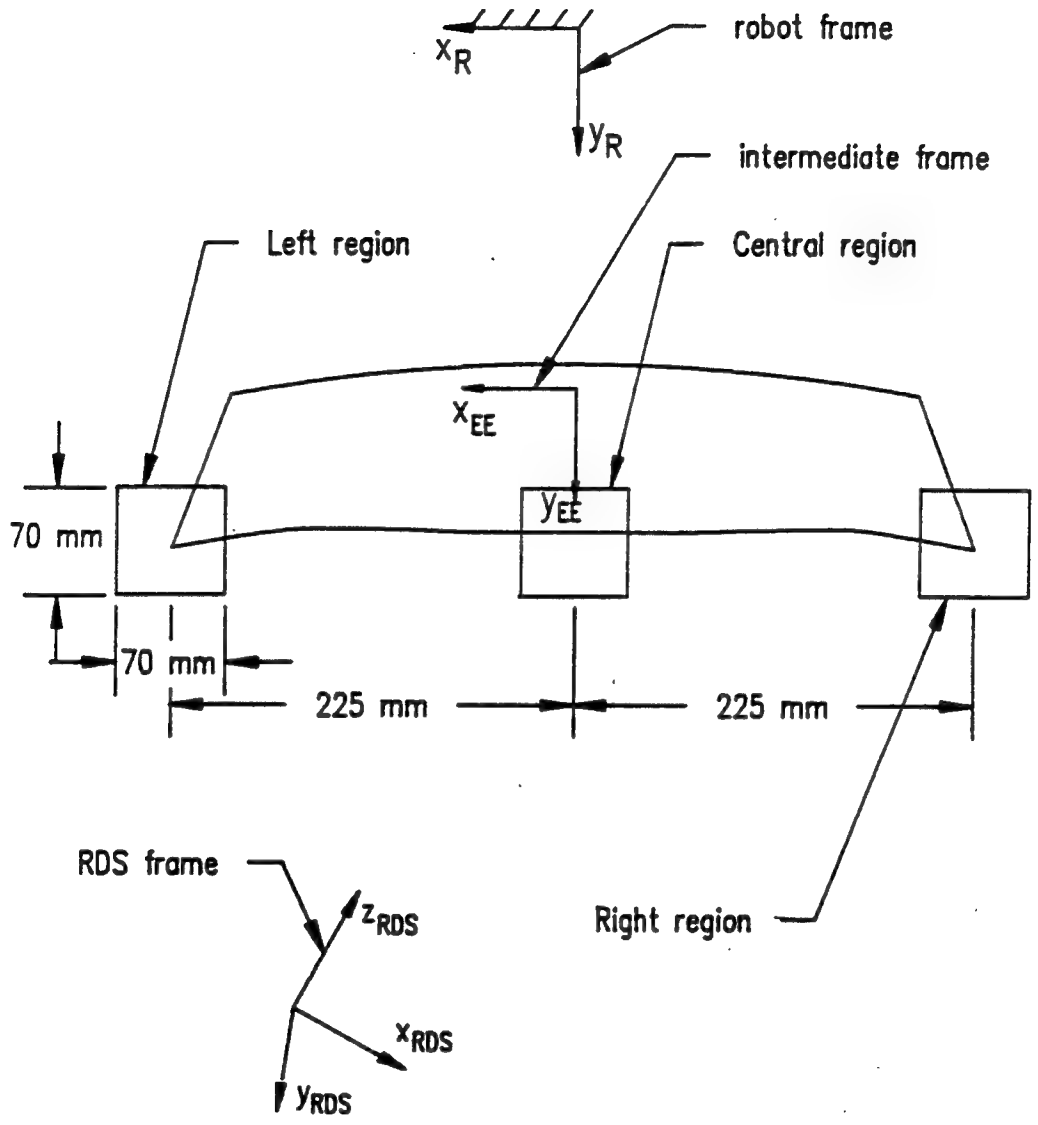


Figure 4.2. Left, Central, and Right common point regions.

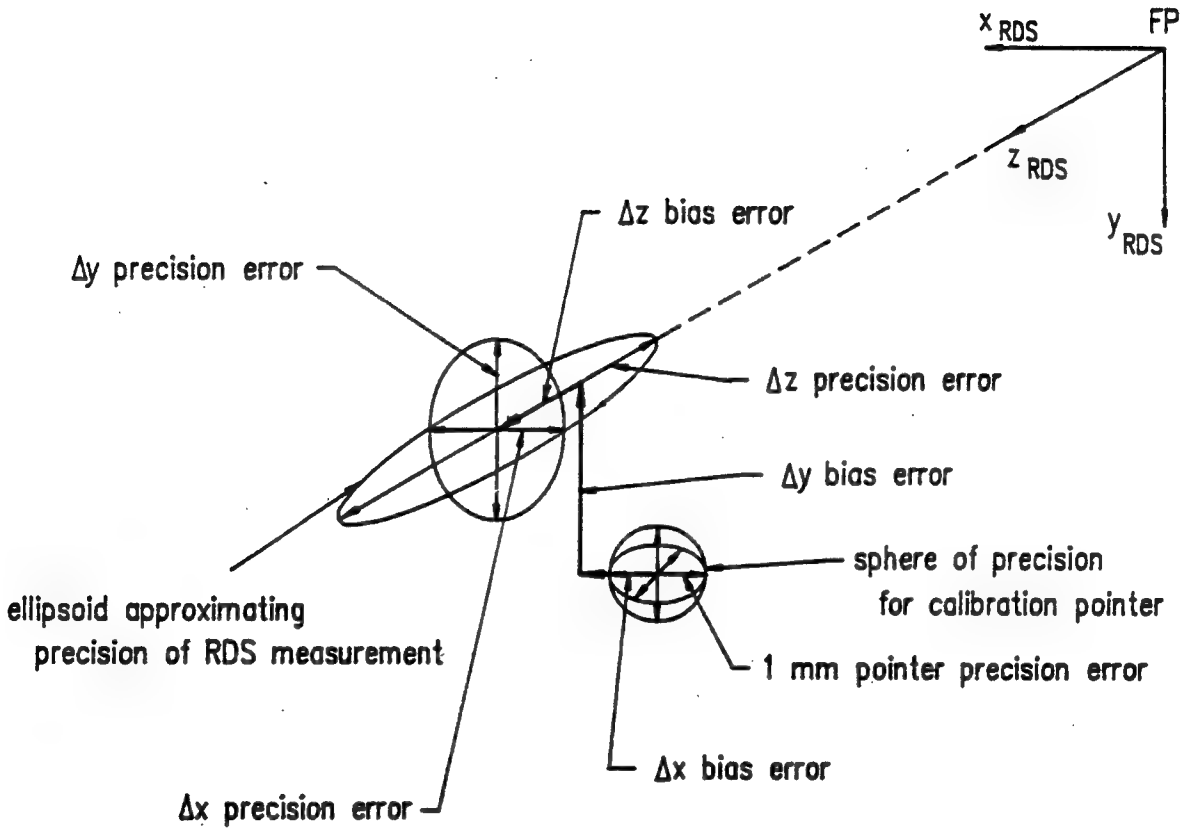


Figure 4.3. Three-Dimensional representation of bias and precision error.

TABLE I

BIAS AND PRECISION ERRORS IN MEASUREMENT
FOR COMMON POINTS MEASURED BY BOTH RDS
AND ROBOT CALIBRATION POINTER

Region	Δx error mm	Δy error mm	Δz error mm
LEFT 8 samples bias:	3.38	1.58	-5.58
precision:	1.35	0.66	2.24
CENTRAL 8 samples bias:	0.62	2.35	-1.55
precision:	0.46	0.71	1.57
RIGHT 8 samples bias:	-4.19	3.95	2.16
precision:	1.07	0.61	2.45

The component bias errors are smallest for the central region, which is close to the intermediate frame location (EE_frame) used for RDS calibration. The smaller bias errors of less than 3 mm indicate that the calibration is most accurate in the center of the image frame, which corresponds to the central region. Two sources of error are present which explain the inconsistency in bias error: (1) camera lens aberrations, and (2) inaccurate RDS calibration. Lens aberrations distort the image with the barrel effect [19] to give the poorest image replication about the image perimeter. Calibration error, on the other hand, involves an inaccurate transformation matrix used to convert RDS to robot coordinates.

The component bias errors ($\Delta x, \Delta y, \Delta z$) were examined for trends to evaluate a cause for the measurement error. For each of the component bias errors, a pattern exists where the bias error values of the left and right region bound the central region bias error. Additionally, the bias error for the central region is approximately the average of the other bias errors. For example, the Δz bias errors for the left and right sides are -5.58 and 2.16 mm respectively, and average -1.71 mm. The Δz bias error of the central region is -1.55 mm, which is close to the -1.71 mm average. A correlation is therefore exhibited which can be explained by a skew in the orientation of the calibration transformation matrix, implying that inaccurate calibration is primarily responsible for the bias errors. Since lens aberration effects are non-

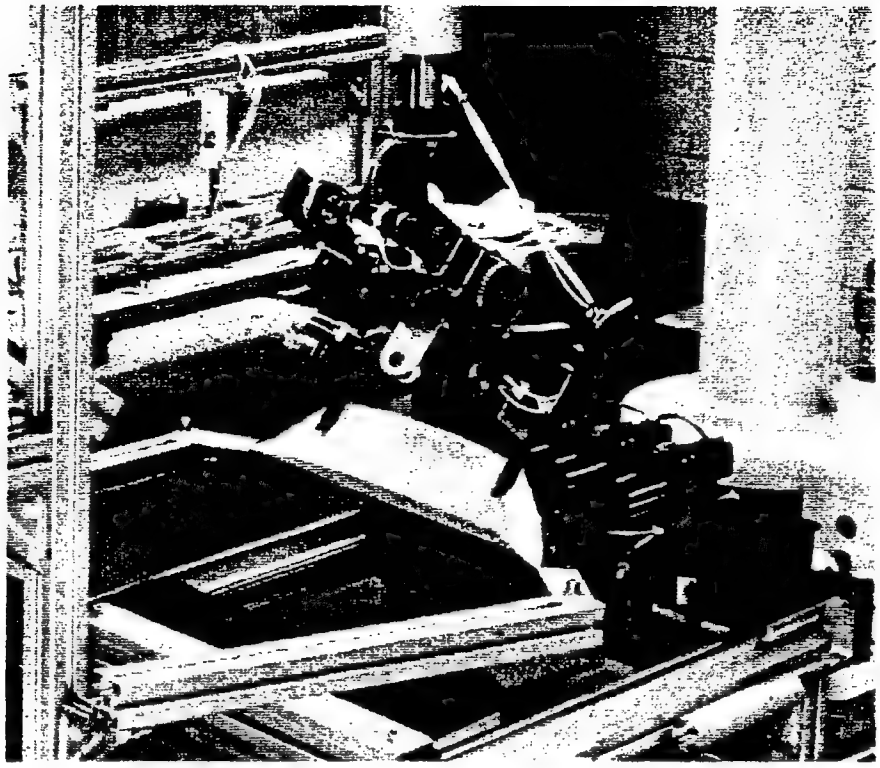
linear, they are not the dominant cause for the bias error in these measurements.

The relatively low precision errors indicate that the system is repeatable, and could be tuned through further calibration to yield improved results. The precision errors ranged between 0.46 and 2.45 mm, with the measurement along the depth axis (or Δz error) the least precise. This is expected from a system that uses a 2-D camera for depth perception.

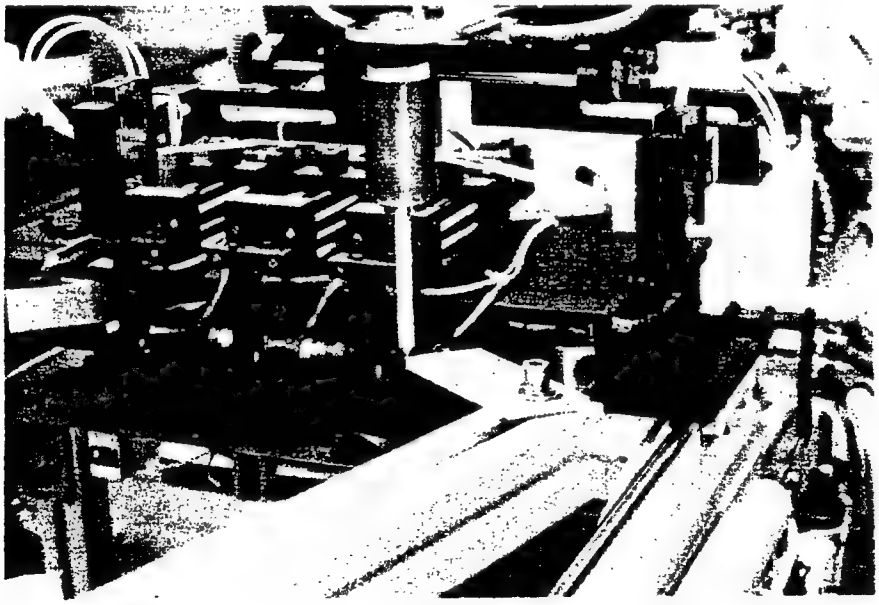
Collar Model Accuracy

The accuracy of the collar model is partially dependent on RDS accuracy to give good range data measurements for the estimates of the collar points. Thus, there is a propagation of errors, or an accumulation of error, in the collar model since the collar model algorithm utilizes RDS measurements. The collar model accuracy also depends on how well the collar model algorithm predicts the collar point locations with the line striping and image processing procedures.

Eight collars were cycled through the workstation to test for collar model accuracy. Since line striping and collar model generation occur while the collar is in the presentation position shown in Figure 4.4(a), which corresponds to the robot position loc[12], the locations of the collar points were marked with an external device designed to physically identify the collar points before the collar was moved to the next position. This device is shown in Figure 4.4. The workstation cycle was discontinued following collar



(a) device marks the location
of the presented collar points



(b) calibration pointer positioned
to measure the marked location

Figure 4.4. Device for physically identifying presented collar points.

placement on the vacuum surface to preserve the collar point locations on the surface.

The calibration pointer was used to obtain reference measurements for the collar points by pointing to the marked locations designating the previous collar point positions as shown in Figure 4.4(b). The measured collar point locations were used to determine the ID_frame location, which is located equidistant between the collar points. A comparison between the collar model predicted and the robot measured ID_frames was conducted similar to the analysis for RDS accuracy. Bias and precision errors were evaluated for an estimate of collar model accuracy.

A visual means for studying the collar model error is also available by viewing the camera monitor in addition to the physical measurement errors. Striping, filtering, and region growing is seen on the monitor as it occurs. Following the striping (three stripes) for each point, a small circle is drawn on the monitor at the 2-D location of the estimated collar point corresponding to left_point$(x,y)_i$, or right_point$(x,y)_i$, as depicted in Figure 4.5. By comparing the location of the circle center with the image of the collar point, the viewer can qualitatively evaluate the performance of the collar model algorithm. Figure 4.6(a) shows a collar model image as seen on the monitor, and Figure 4.6(b) shows the model superimposed on the original collar image. Qualitatively, the collar model favorably predicts the collar point locations.

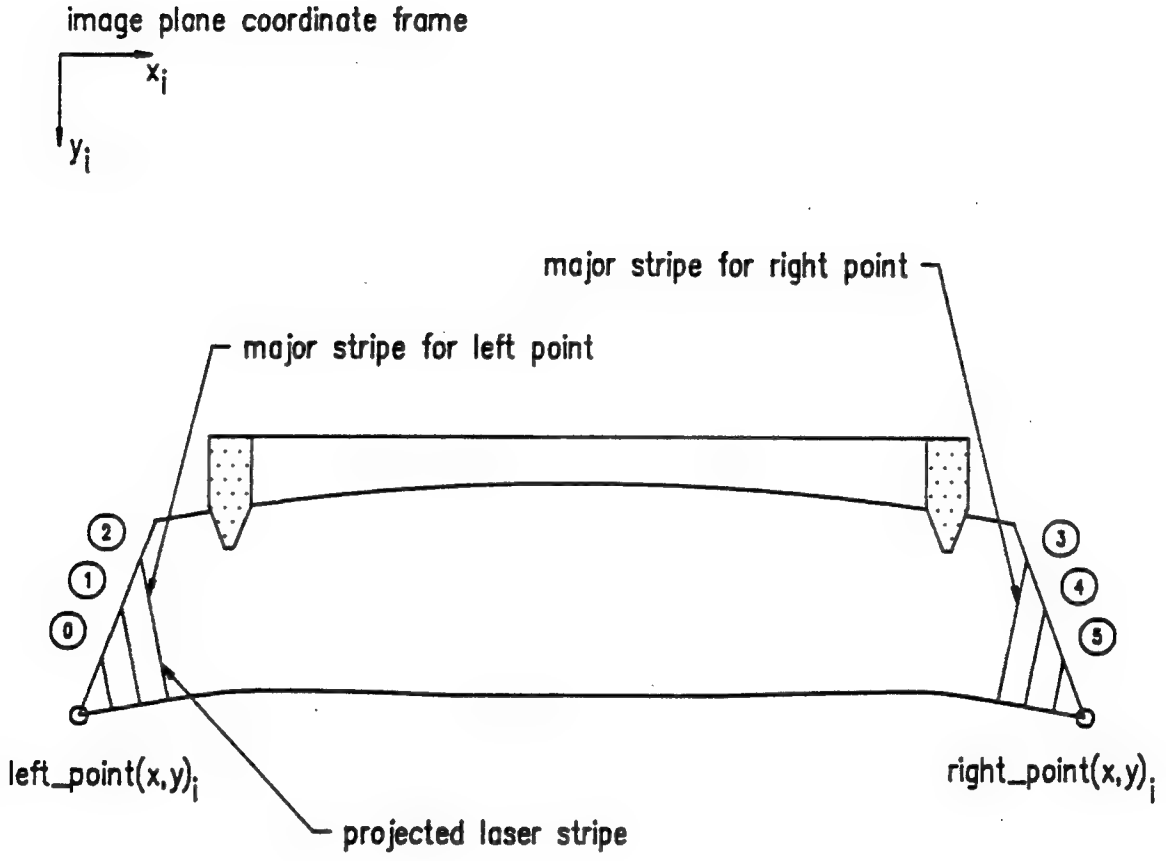
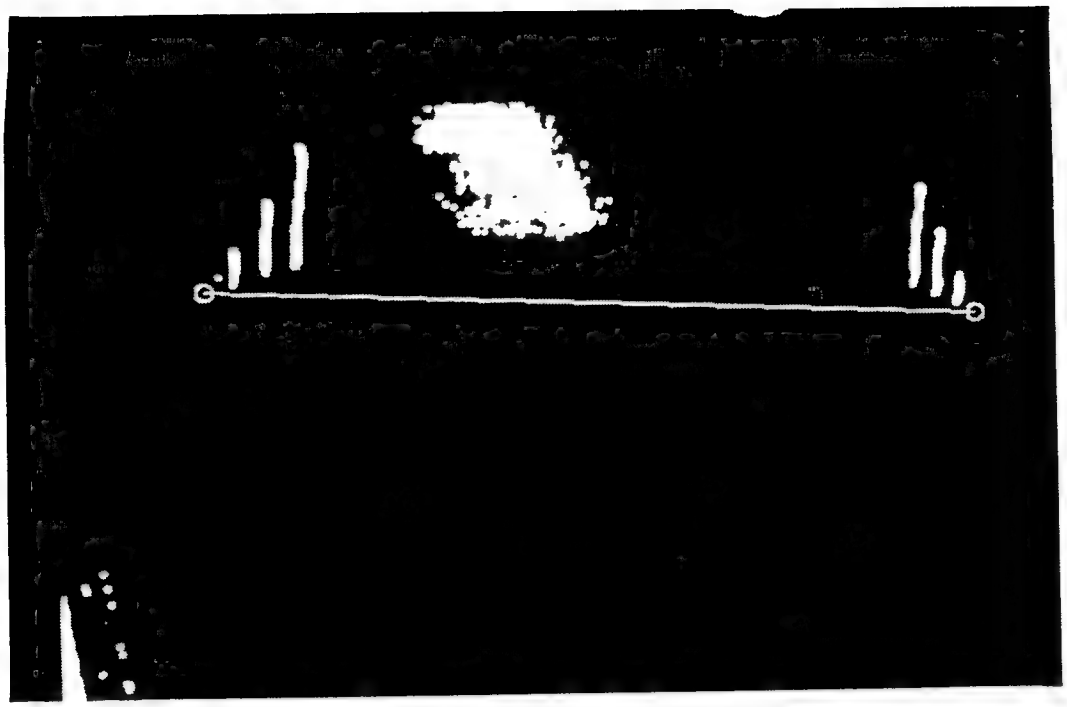
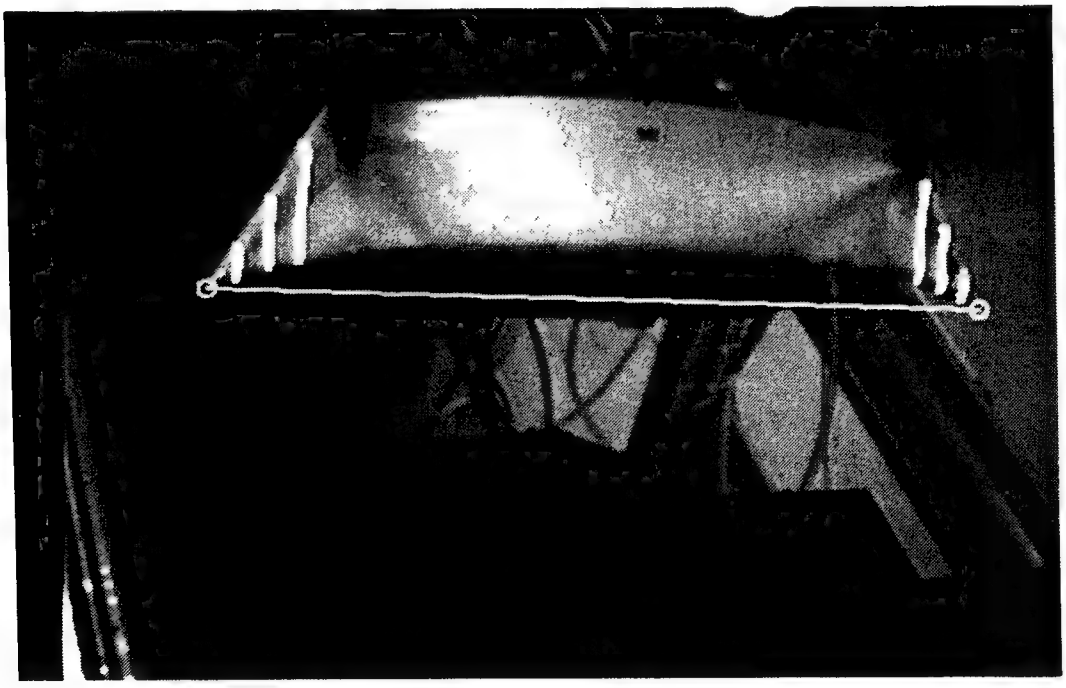


Figure 4.5. Two-Dimensional left_point(x,y)$_i$ and right_point(x,y)$_i$ estimated by collar model algorithm.



(a) collar point estimates generated by the collar model algorithm



(b) collar model superimposed on original collar image

Figure 4.6. Example of collar model.

Table II presents the bias and precision errors determined by the comparison between collar model predicted and robot measured ID_frames for the eight collars sampled. The error values were obtained with Equations 4.1-4.3. The errors listed in Table II are due in part to the propagation of RDS calibration and pointer measurement errors, and the remainder is due to error in the collar model. The collar model component of error is a direct result of the collar point 2-D estimates, $left\_point(x,y)_i$ and $right\_point(x,y)_i$, and the extrapolation for the mirror angles (θ) necessary to obtain 3-D estimates for the collar points.

TABLE II

BIAS AND PRECISION ERRORS IN THE MEASUREMENT OF ID_frame MEASURED BY BOTH THE COLLAR MODEL ALGORITHM (RDS) AND THE ROBOT CALIBRATION POINTER

Region	Δx error mm	Δy error mm	Δz error mm
ID_frame 8 samples			
bias:	0.27	5.73	2.17
precision:	0.86	0.85	2.02

The Δx bias error is similar to that of the RDS accuracy analysis, both less than 1 mm; but the respective Δy and Δz bias errors are both different by approximately 3.5 mm. This difference is attributed to the collar model point estimates. Figure 4.7 gives an example of a collar model superimposed on the original collar image which illustrates the inaccuracy in the algorithm. The point estimates do not comply with the actual collar points. These inaccurate estimates are responsible for the Δy and Δz bias errors; there is no resulting Δx bias error since the individual collar point Δx bias errors are in opposing directions and are averaged to a small value. This trend displayed in Figure 4.7 was observed for a majority of the eight sampled collars, and is caused by several stripe endpoints that do not coincide with the collar contour, which in effect, yields a regression intersection that does not match the collar point.

The precision errors for Δx and Δy were less than 1 mm, and was 2 mm for Δz suggesting that the collar model algorithm is repeatable in its collar point estimations. This repeatability indicates that the 3-D collar point estimations could be tuned with compensating offsets equal to the component bias errors in the respective component directions to yield a consistently better collar model accuracy. However, prior to compensating for bias errors or symptoms, the source of the problem involving the stripe endpoint approximation of the collar contour should be improved. Due to the limited set of data taken, the reported ranges for precision error

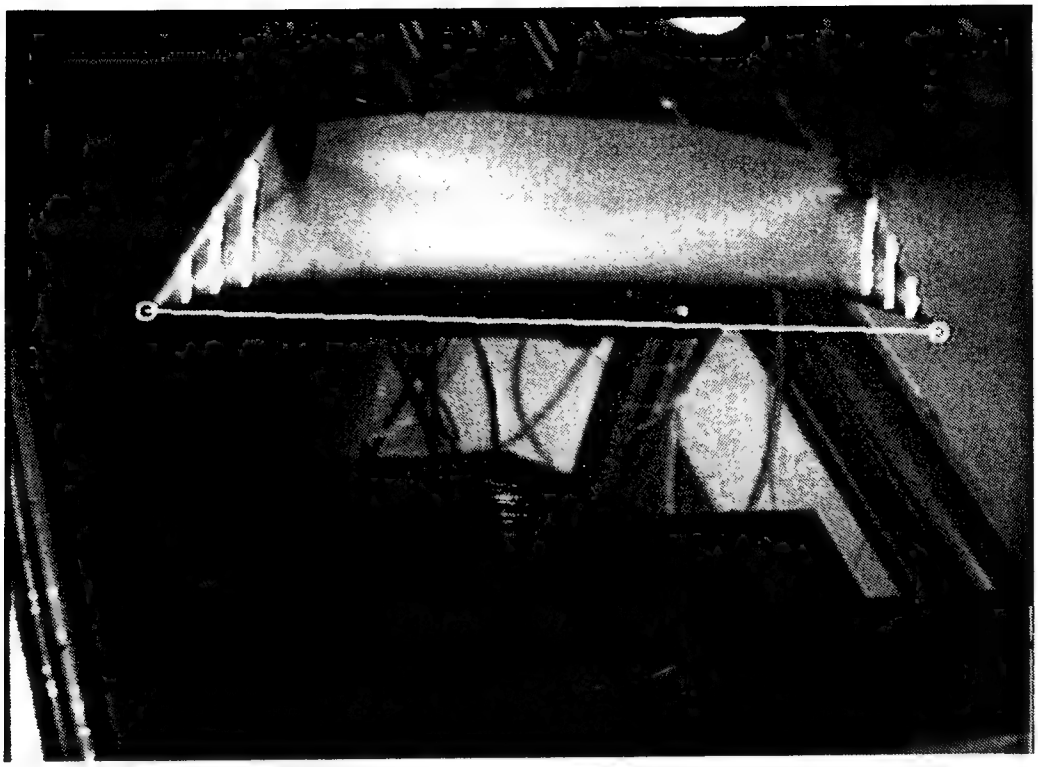


Figure 4.7. A collar model which demonstrates inaccurate point prediction.

refer to a confidence level of approximately 75%. This confidence level reflects a limited statistical basis for specifying system performance. The use of student t-distribution statistics suggests that twenty or more samples are required to attain a 95% confidence level.

Collar Loading Accuracy

The collar is loaded on the vacuum surface with the collar points located to conform to the fixed geometry of the creaser blades to be inserted and extended. The collar point locations for eight collars were measured with the robot calibration pointer subsequent to being loaded on the vacuum surface as shown in Figure 4.8. The robot pointer measurements were compared with the target positions for the collar points which coincide with the creaser blade tip locations when the blades are in the inserted and extended position.

The target points serve as the reference locations to which the robot calibration pointer measured locations are compared. Collar placement on the vacuum surface is variable in the x_R and y_R dimensions; the z_R dimension is constrained by the vacuum surface. The component errors are evaluated for two dimensions per point using

$$\Delta x = x_{collar\_point} - x_{blade\_tip}, \text{ and} \tag{4.7}$$

$$\Delta y = y_{collar\_point} - y_{blade\_tip}, \tag{4.8}$$

giving the error an orientation with respect to the robot coordinate system. Additionally a magnitude error is evaluated with

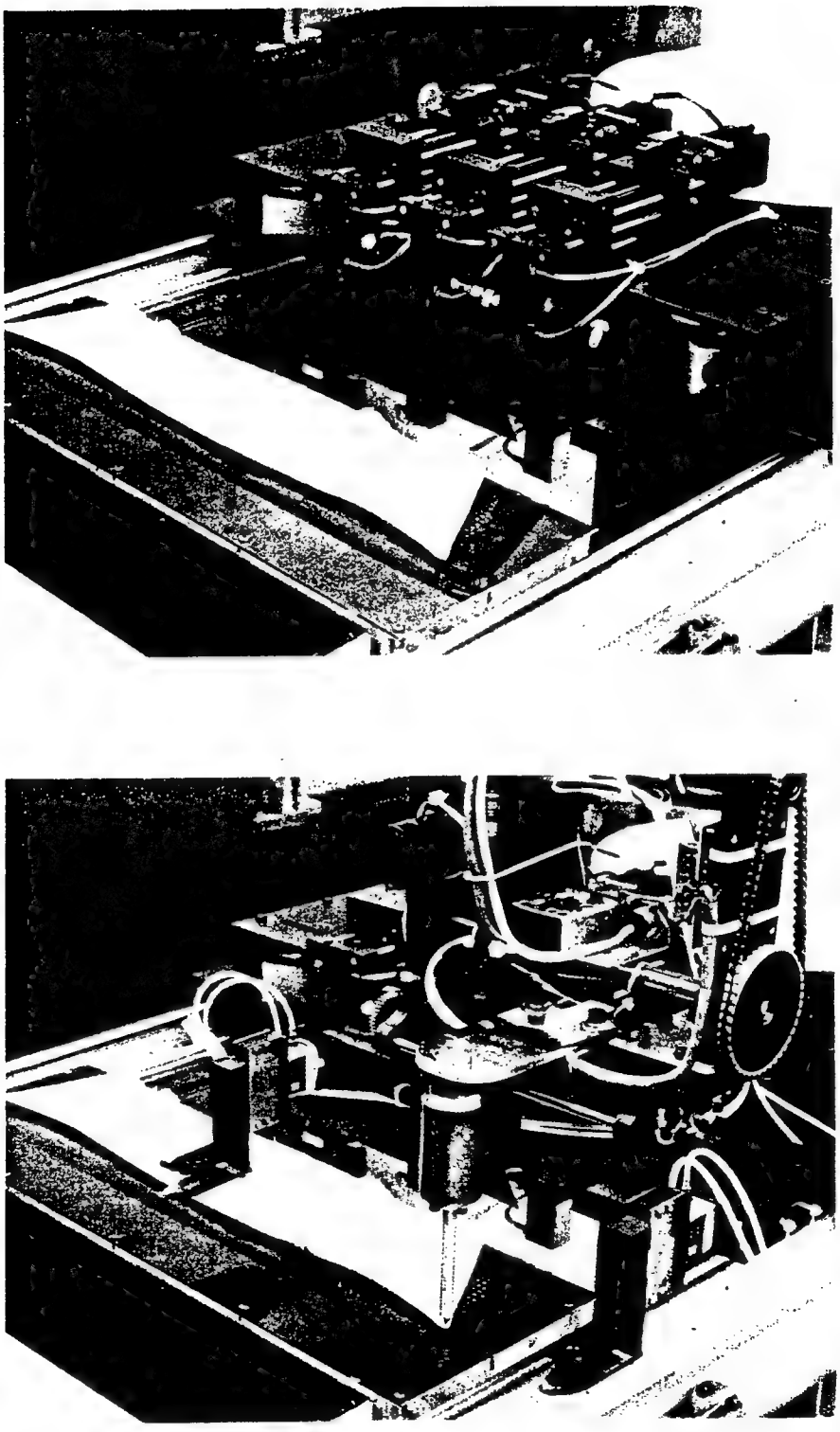


Figure 4.8. Collar positioned on vacuum surface.

$$E = |\sqrt{\Delta x^2+\Delta y^2}| \tag{4.9}$$

for the measured difference between collar point and respective blade tip.

Table III summarizes the bias and precision errors measured between collar points located on the vacuum surface and their target positions. All eight collars were good candidates for creaser blade insertion, suggesting that the pressing machine is capable of handling a placement tolerance exceeding 5 mm, the precision for the magnitude error E. The Δx and Δy bias errors represent errors from two sources: (1) the collar model algorithm, whose frame match introduces error caused by an inaccurate ID_frame assignment, and (2) the friction which exists between the unmodeled bottom ply and the vacuum surface when the collar first makes contact prior to the vacuum draw. There is also a component of measurement error in the measured creaser blade tip locations owing to the calibration pointer. Figure 4.9 illustrates the average loaded collar position in reference to the target location. The average position is established by the component bias errors. The resulting collar skew is due to a bias error in the $\Delta\theta$ component of the error vector DELTA-$(\Delta x,\Delta y,\Delta\theta)$, which is a function of the modeled collar point locations. It is important to note that as the creaser blades are extended in the collar pocket, they self align the collar and collar points to conform to the fixed geometry of the pressing workstation. This results in a correctly positioned collar for edge creasing and pressing.

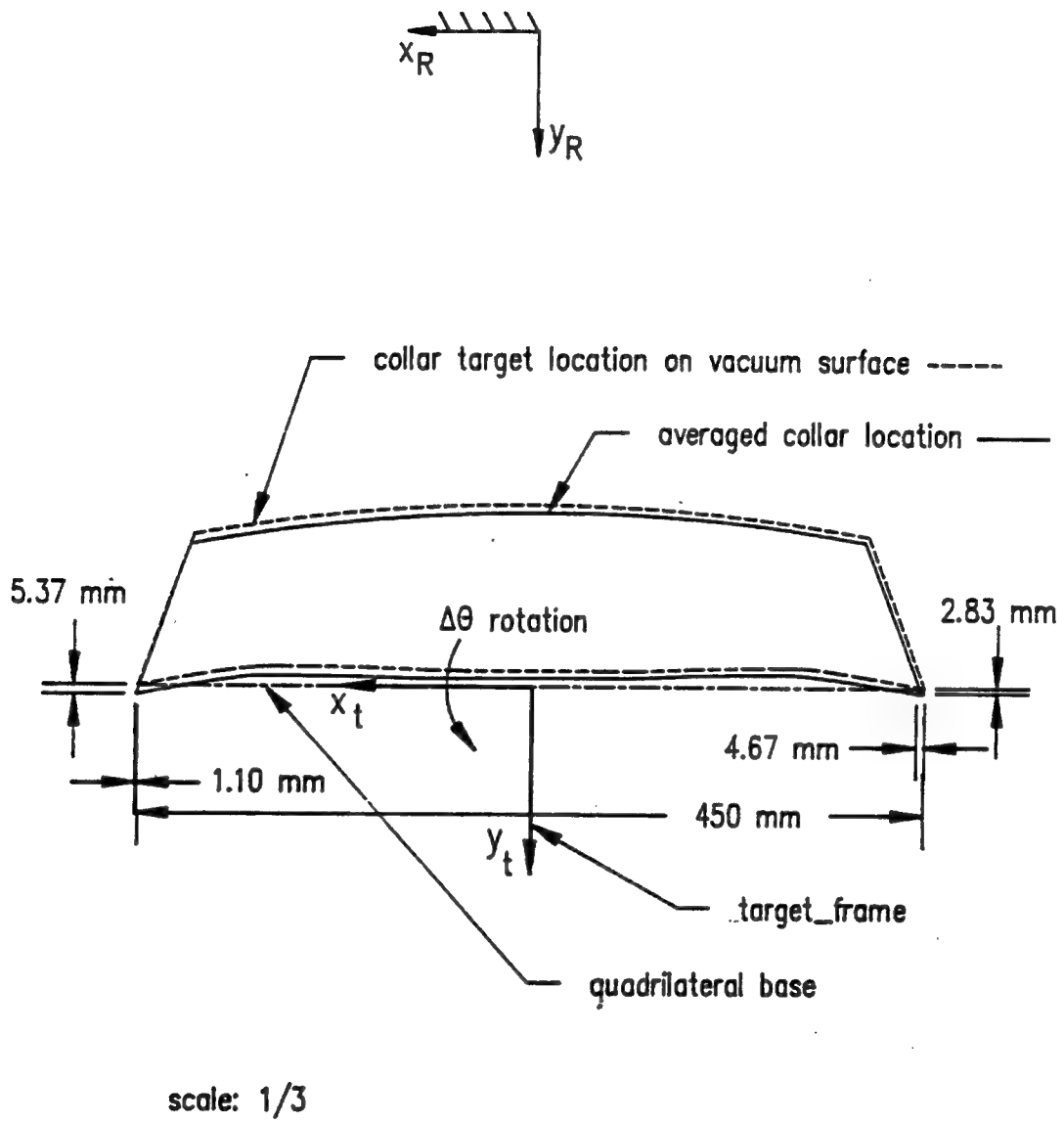


Figure 4.9. Average loaded collar position versus target position.

TABLE III

BIAS AND PRECISION ERRORS FOR THE MEASUREMENT BETWEEN LOADED COLLAR POINTS AND RESPECTIVE EXTENDED CREASER BLADE TIP POSITIONS

Region	Δx error mm	Δy error mm	ΔE error mm
LEFT 8 samples			
bias:	1.10	5.37	5.46
precision:	0.85	3.44	3.82
RIGHT 8 samples			
bias:	4.67	2.83	5.21
precision:	1.18	2.11	4.31

System Speed

System speed is integral to successful automation. The collar loading operation is divided into six distinct time segments for a total process time of approximately 22 seconds dependent on the robot speed. The computational efforts of line striping, collar model generation, and error vector passing, which account for the VC controlled contribution to collar loading, are performed in 16 to 18 seconds of the 22 second time sequence.

The physical line striping operation is controlled by the System Supervisor in parallel with the vision processing, and does not contribute to the time involved in generating the collar model. The image processing for each stripe takes slightly less than 2.5 seconds. Of the 2.5 seconds processing time, region growing accounts for 1.5 seconds, the linescan and edgefind bug algorithm takes between 0.6 and 0.8 seconds, and the computation time for diagnosing the stripe requires the remaining time. Thresholding is performed by a look up table (LUT) operation in the vision hardware that occurs once during every vertical sync signal to the monitor. This operation requires no processing time. At six stripes per model, and nearly 2.5 seconds per stripe, less than 15 seconds is required to acquire and compile the stripe information for the model. The more than one second of remaining time is used to evaluate the collar point locations and pass an error vector to the SS.

The message passing is conducted at 1500 characters per second (cps), using less than 0.5 seconds for the error vector to pass between the VC to the RC via the SS. The robot trajectory generation also requires less than 0.5 seconds. The remaining process time is consumed by the robot motions required to load the collar. Since the robot speed is specified by the operator, the total time for collar loading with the existing system using velocities of 20 ips would take about 20 seconds.

System Operation

The fully automated AAW workstation is designed to accept bundles of stacked unturned collars and process them individually with a turning and pressing operation. At present, the process time for the AAW to retrieve, turn, and prepare an individual collar for pressing is 1 minute and 35 seconds. The time to accomplish the same task for an experienced operator with a manually operated pressing machine is 45 seconds. The AAW process time, however, is hampered by temporary "wait" and "pause" statements in the control code, and has not yet been improved with faster robot, end-effector, or workstation device speeds. The system has the potential for speeds comparable to that of the human operator. A videotape is available that demonstrates the functional operation for the complete AAW system. The 3-D sensing of the turned collar is presented along with the visual activity displayed on the monitor during the process.

CHAPTER V

CONCLUSIONS AND RECOMMENDATIONS

Conclusions

A 3-D vision range finder has been implemented in a workstation dedicated for turning and pressing shirt collars to demonstrate its usefulness for apparel material manipulation. The range finder system, or Range Data Scanner (RDS), has been developed to perform three-dimensional vision sensing or depth location of the collar geometry for creating a geometrical collar model for material handling. The collar model is used to generate the robot motion trajectory for properly positioning the collar on a pressing work surface.

The system is successful in loading collars within a range acceptable for proper creaser blade insertion on the pressing workstation. The vision sensor compensates for collars which are retrieved from the turner skewed and offset relative to the end-effector by as much as ±3 degrees, indicating that the wireframe quadrilateral collar model is effective and adequate for collar placement. The collar is loaded on the vacuum surface with the points accurate to within 5 mm of their target locations, and a precision of 4 mm, which is within the 11 mm tolerance available between the collar opening and the creaser blade assembly to be inserted.

These accuracy and precision values encompass the errors propagated by both the Range Data Scanner and the collar model algorithm, which are both critical in estimating the collar geometry for placing the collar on the vacuum surface. The precision for the collar model, which includes the propagated RDS precision error, is approximately 2 mm, implying that the system is repeatable. The difference in collar model and collar loading precision is due primarily to the unmodeled bottom collar ply and its interaction with the vacuum surface.

The speed of the collar locating and positioning process is approximately twenty-two seconds and represents the capability of the vision equipment used. With an increased robot speed of 20 ips, this time is reduced to about twenty seconds. Twenty seconds is comparable to the time taken by a human operator to perform the same function in a repeatable and consistent way. Thus, the potential for 3-D vision sensing for applications in apparel assembly is demonstrated.

It is important to recognize that the AAW will perform collar loading satisfactorily without 3-D vision sensor information. This is due to the regularity in collar retrieval by the robot from the turning machine, and proved by the statistical analysis of the collar position measurements acquired by both the robot and RDS. Without the 3-D vision sensing operation, collar turning and pressing can be performed in less than 1 min. 15 sec. by the AAW.

This research has demonstrated the potential for using three dimensional vision sensing for flexible material handling and shown that 3-D vision sensing is a viable technology for the apparel industry. The currently designed 3-D vision sensing system is not required for direct implementation of the AAW on the factory floor. However, the application demonstrates that an apparel research knowledge base has been developed for 3-D vision sensing which is available for other advanced or appropriate factory floor applications. An example of such a system is Nakamura's 3-D sewing system which uses stereo vision to guide a compact sewing machine to sew fabrics in a 3-D workspace [23]. Domey [15] also uses 3-D vision sensing to enter object geometric data into a CAD system to create a surface profile database for the design and inspection of garments.

Recommendations

System speed requires the most improvement for application of this system in an industrial setting. The speed could be improved with a VME-based computer workstation, which has an architecture more suited for real-time control applications as compared to the SS microcomputer. The Adept MC1 controller might also be employed for this purpose since it is a workstation controller, and is not limited to robot control. Difficulty would be involved, however, in integrating the versatile Data Translation vision system with the MC1 controller. A PC would be required, involving the communication burden that exists with the SS computer. Image

processing the individual stripes with parallel processing would significantly reduce the time taken by the vision process.

The system accuracy is satisfactory for this application, however, other applications such as guiding a sewing head in a 3-D workspace will require better accuracy. Bias error in the range data measurements could be improved with further attention to the calibration of the camera in workstation coordinates. The calibration approach used for this research, as described in Appendix C, could be implemented several times to obtain a statistical sample leading to a "best fit" transformation matrix. Another approach would consider localized calibrations corresponding to mapped regions in the workspace, with individual calibrations for each collar point region that effectively eliminate the respective bias errors. A calibration method which does not involve the end-effector pitch motion would reduce the calibration pointer precision error of ±0.5 mm (±20 mil) to the robot precision error of ±1/40 mm (±1 mil); a pointer directly attached to the robot wrist would provide this improved precision.

The collar model for estimating collar point locations can be improved using more stripe data per collar point. Additional data would enhance the collar model linear regression analyses, which in turn may improve the collar point contour and give a better approximation for the contour intersection. Increased data also permits higher order regressions for the collar point contours which is

appropriate since the collar point contours are not linear. By projecting more stripes and involving more complex computation, however, the time used for generating the collar model geometry is increased. Sato's method of space encoded range imaging could be utilized to gain considerably more range data per period of time [12], and perhaps evaluate the collar points faster.

The driving factors in designing manufacturing equipment for the apparel industry must be to keep it simple and low-cost. Sophisticated vision capabilities for automating manufacture of apparel may not be cost effective, but it may be prudent to design apparel assembly processes with some level of vision capability when possible. Binary vision and proximity sensor devices are available which could provide enhanced machine performance capabilities.

Numerous opportunities exist for future research in vision sensing of flexible apparel materials. For example, more advanced mathematical models for apparel workpieces are necessary to understand the fabric surface profile and its behavior for robotic handling and processing. Another area involves real-time control for fabric manipulation with visual feedback. The research project for guiding a sewing head about 3-D seam contours, which is parallel to vision guided welding, will establish both real-time machine vision strategies and modern control techniques for the apparel community.

APPENDICES

Appendix A

Equipment Specifications

Range Data Scanner Components

HeNe Laser

Manufacturer: Aerotech Inc.
101 Zeta Drive
Pittsburgh, PA 15238
(412) 963-7459

Part Number: HeNe 210R 200 Series Laser

Description: 1 mW randomly polarized laser head with flying leads, 12 VDC, 0.64 mm beam diameter, 1.27 mrad beam divergence, 632.8 nm wavelength.

Line Generator

Manufacturer: Aerotech Inc.

Part Number: LG-1

Description: Cylindrical lens to produce line stripe.

HeNe Laser Mirror

Manufacturer: Edmund Scientific
101 East Glouchester Pike
Barrington, NJ 08007-1380
(609) 573-6250

Part Number: B31496

Description: First surface mirror, 47x51 mm.

Timing Belt and Pulleys

Manufacturer: Winfred M. Berg, Inc.
499 Ocean Avenue
East Rockaway, NY 11518
(516) 599-5010

Part Numbers: 20TB-80, 20TP8-12, 20TP6-48

Description: Timing belt and pulleys, 1:4 ratio.

Stepper Motor and Driver

Manufacturer: Oriental Motor USA, Corp.
2701 Plaza Del Amo, Suite 702
Torrance, CA 90503
(213) 515-2264

Part Number: UMD 245-AA

Description: SUPER VEXTA UMD Step Motor/Driver Package.

Controller Board

Manufacturer: Precision Micro Control Corp.
3555 Aero Court
San Diego, CA 92123
(619) 565-1500

Part Number: DCX

Description: Eight-axis motion controller board.

Stepper Motor Module

Manufacturer: Precision Micro Control Corp.

Part Number: DCX-MC150

Description: Plug-in module used for step motor.

Proximity Switch

Manufacturer: efector, Inc.
805 Springdale Drive, Whiteland Business Park
Exton, PA 19341
(800) 441-8246

Part Number: IF3002BPKG

Description: Solid state proximity switch.

Vision Hardware Specifications

CCD Camera

Manufacturer: Panasonic Industrial Company
One Panasonic Way
Secaucus, NJ 07094
(404) 368-0160

Part Number: WV-CD20

Description: 510(H)x492(V) Element CCD Type; 6.6x8.8mm^2 scanning area equivalent to a 2/3" pick-up tube.

Automatic Iris Lens

Manufacturer: Panasonic Industrial Company

Part Number: WV-LA16B

Description: Auto iris lens, 16mm focal length.

Arithmetic Frame Grabber Board

Manufacturer: Data Translation, Inc.
100 Locke Drive
Marlboro, MA 01752-1192
(508) 481-3700

Part Number: DT2861

Description: 512x512 frame-store memory buffers, 8 bit resolution.

Auxiliary Frame Processor Board

Manufacturer: Data Translation, Inc.

Part Number: DT2858

Description: Provides features such as graphic overlays.

Eight-Channel Video Multiplexer Board

Manufacturer: Data Translation, Inc.

Part Number: DT2859

Description: Permits eight video sources.

Computers, Robot, and Controller

VC and SS computers

Manufacturer: Advanced Logic Research (ALR), Inc.
9401 Jeronimo
Irvine, CA 92718
(412) 963-7459

Part Number: FlexCache 25386 computer

Description: IBM-type computer, 25 MHz, 1 MByte RAM.

Robot and Robot Controller

AdeptOne Robot

Manufacturer: Adept Technology, Inc.
150 Rose Orchard Way
San Jose, CA 95134
(408) 432-0888

Part Number: AdeptOne Robot

Description: Four-axis SCARA configuration robot.

Adept MC Controller

Manufacturer: Adept Technology, Inc.

Part Number: Adept MC Controller

Description: Motorola 68000 based workcell controller.

Appendix B

Range Data Calibration

Pinhole Camera Model

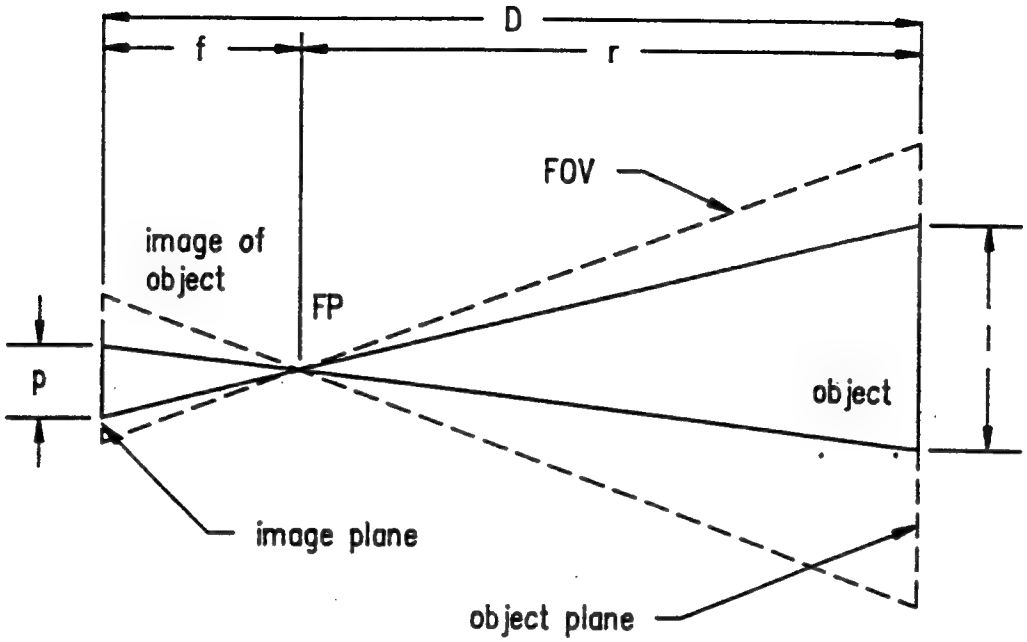
Calibrating the RDS requires modeling the Panasonic camera with a pinhole camera model. The pinhole camera model geometry is shown in two dimensions in Figure B.1. This geometry is appropriate for either the top or side views of the camera, corresponding to the pixel distance in either the x_i or y_i directions, respectively. Modeling the camera consists of determining the focal length f, which is the distance between the focal point FP and the image plane. The image plane is an array of 512 x 512 pixels with dimensions of 8.8 mm x 6.6 mm corresponding to a pixel width (in the x_i direction) of 0.15 mm and pixel height (in the y_i direction) of 0.12 mm. By comparing the acquired image with the object image dimensions, a value for the focal length is determined.

According to the manufacturer specifications, the image plane is physically located 15.7 mm from the face of the lens mount as shown in Figure B.1. Since this distance is known, the distance D between the image plane and the object surface can be measured. The following geometry is used with experimental measurements to derive the focal length f for the camera. The distance D is composed of the range distance between FP and the object surface, and the focal length as

$$D = f + r. \tag{B.1}$$

By similar triangles,

$$p/f = l/r. \tag{B.2}$$



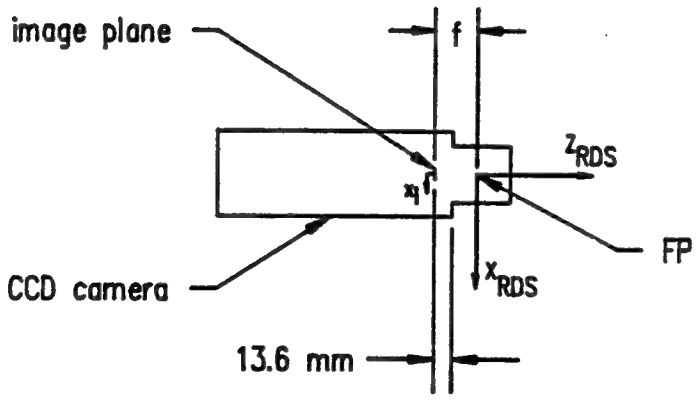


Figure B.1. Pinhole Camera Model Geometry.

By substituting (B.1) in (B.2), an equation with one unknown, the focal length f, is

$$p/f = 1/(D-f). \tag{B.3}$$

Rearranged, an equation for f is derived as

$$f = p \cdot D/(1+p). \tag{B.4}$$

By numerous experiments comparing the image pixel length with the object surface length at distances (D) near one meter, the focal length is estimated as 17.0 mm for the x_i direction and 16.75 mm for the y_i direction.

RDS Geometry

Figure B.2 shows the RDS geometry used for evaluating range data. Since triangulation requires two angles and a base length, the triangle corner points M and FP on the RDS platform are critical for accurate range data. Point M represents the point of reflection on the mirror, which is located with respect to the RDS platform as per the blueprint design tolerance of ±2 mil. Point FP represents the focal point of the camera as determined relative to the camera in the previous section.

Point FP affects the values for d, α, and γ, and is dependant on the camera position relative to the RDS platform. Though the camera position can be approximated, its precise position relative to the RDS platform cannot easily be measured. This is due to loose tolerances and a lack of specifications for the camera geometry. The angle of the camera relative to the platform, however, was designed and measured as 23.5 ± 0.1 degrees. Therefore, the axial

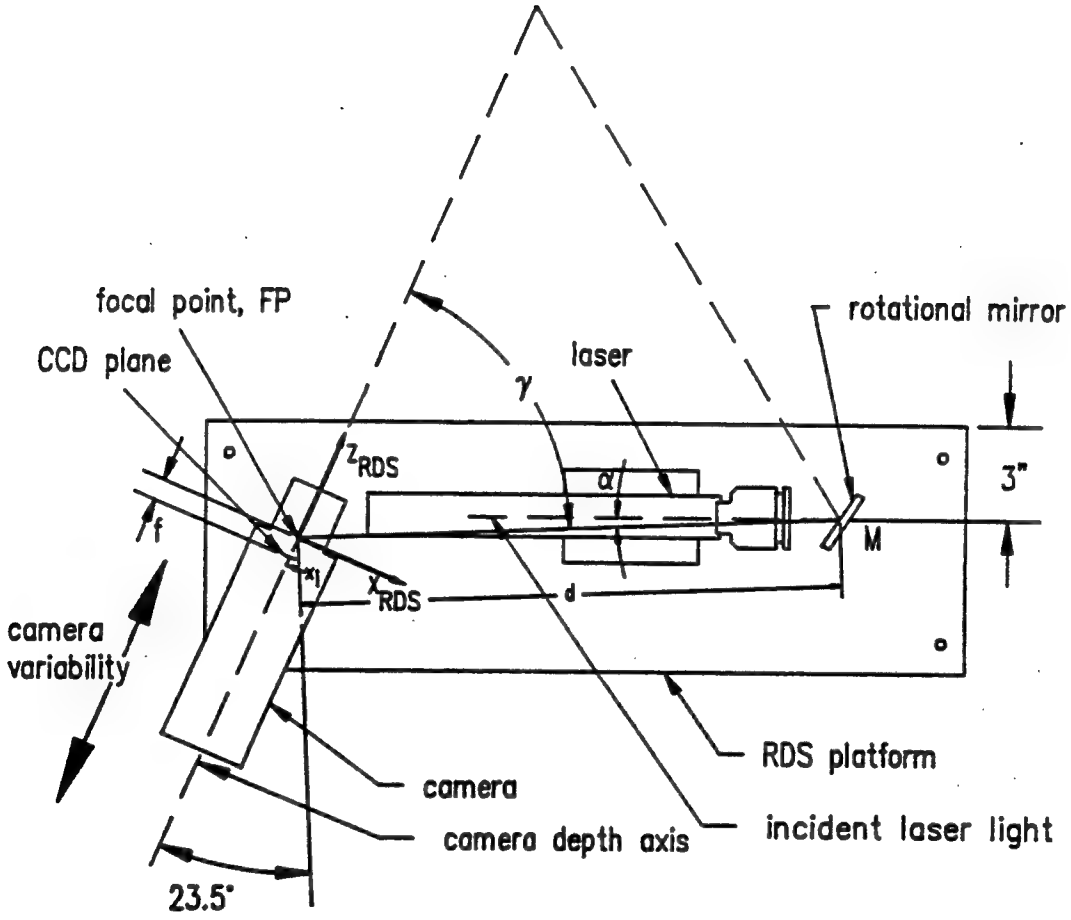


Figure B.2. RDS geometry.

direction and centerline for the camera is known, but the focal point FP remains loosely approximated. Figure B.2 shows a double-headed arrow to indicate the variability in the camera position relative to the platform. The angles for α and γ, and the distance d are therefore dependent on the focal point position FP.

Determining the Camera Position

Stripes projected at a flat surface parallel to the RDS platform were observed with the vision system to determine the FP position analytically. Figure B.3 shows the configuration used for conducting this experimentation. The RDS platform was located a distance w from the flat surface, or wall, to approximate the one meter range of a presented collar. Stripes were projected on the wall with known geometry and observed on the camera image plane by the vision system.

The geometry for the stripes projected on the wall and the resulting stripe locations on the image plane were plotted in an AutoCAD drawing. Using the AutoCAD STRETCH command, the image plane along with the lines of view were stretched along the camera depth axis until the focal point FP was 17.0 mm (f for x_i) from the image plane. The position for point FP satisfying this criteria was used to determine α, γ, and d. The resulting values for these parameters as implemented in the triangulation software were 2.47 degrees, 64.03 degrees, and 17.08 inches respectively. The error in these values is studied in the next section.

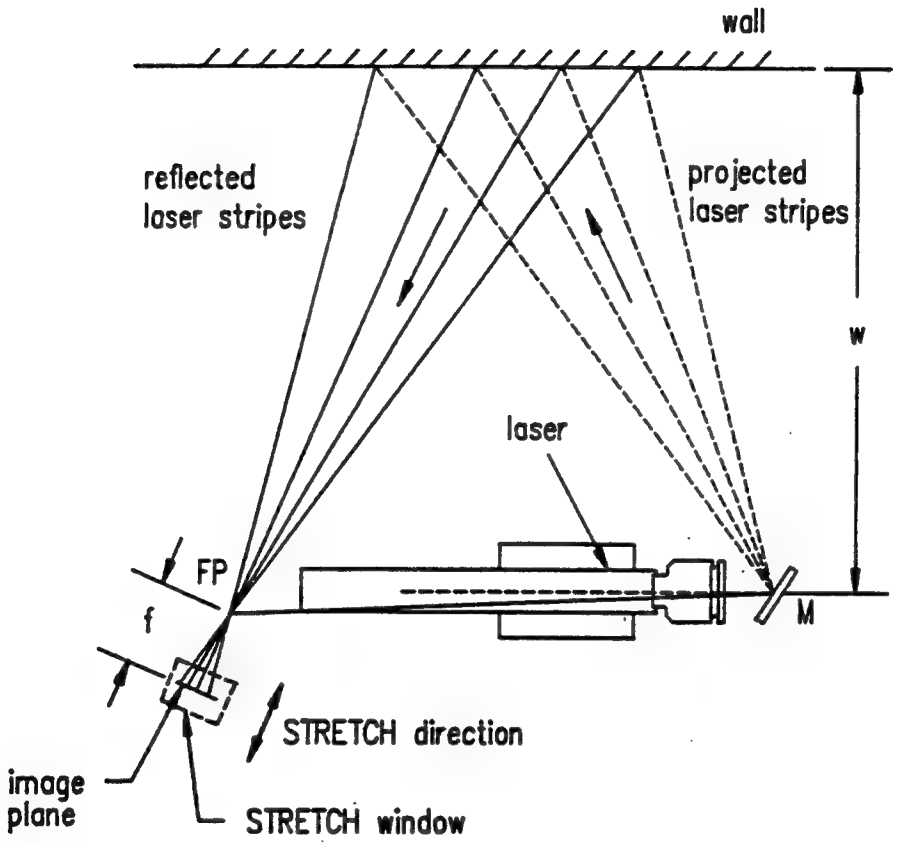


Figure B.3. Configuration for laser striping on wall.

Range Data Uncertainty Analysis

An uncertainty analysis, which is the process of systematically quantifying error estimates, provides a tool for evaluating a measurement system such as the RDS. Each of the parameter values used for RDS triangulation includes a tolerance which introduces an uncertainty error in the resulting range data measurement. Table IV gives each of the parameters used in triangulation and a conservative estimate for the error introduced by each parameter. For each parameter, the resulting RDS measurement error e_i is given by its x_{RDS}, y_{RDS}, and z_{RDS} components. The error e_i encompasses both an accuracy and precision error, as defined in Chapter IV.

TABLE B-I

THE PARAMETERS USED IN RDS TRIANGULATION AND THE CORRESPONDING MEASUREMENT ERRORS

Parameter	Error in Parameter	Component Error of Measurement		
		e_x (mm)	e_y (mm)	e_z (mm)
d	±1.0 mm	0.03	0.11	2.20
α	±5 min.	0.02	0.03	1.22
γ	±15 min.	0.13	0.40	7.95
FL	±0.25 mm	0.21	0.69	0.43
x_i	±1 pixel	0.93	0.10	1.91
y_i	±1 pixel	0.00	0.77	0.00

The component errors for each of the parameters combine to increase the uncertainty of the RDS measurement. A realistic estimate of the combined uncertainty, or propagation of error, for each of the RDS component measurements can be computed using the root-sum-squares method (RSS) by

$$e = \sqrt{\sum_{i=1}^{n} e_i^2}. \tag{B.5}$$

The RSS component errors (e) for an RDS measurement are 0.96, 1.12, and 8.57 mm for the x_{RDS}, y_{RDS}, and z_{RDS} directions respectively. The measurement uncertainty in the z_{RDS} direction is considerably higher than that for the other directions. This is due to the uncertainty in the camera angle γ, which is the most critical parameter in the triangulation computation. The greater uncertainty in the depth direction gives a sense for the difficulty involved in achieving accurate range data. When compared with 2-D visual operations, 3-D operations involve more parameters and greater complexity, and thus, less measurement accuracy.

Appendix C

System Calibration

A conversion between RDS coordinates and robot coordinates is necessary for the collar model to be compared with its target destination in robot coordinates. The RDS must be calibrated with robot coordinates to provide this conversion. RDS calibration is accomplished by locating an intermediate frame with respect to RDS coordinates which is also defined in robot coordinates. Figure C.1 shows the location of the intermediate frame, which is labeled EE_frame to designate a coordinate frame whose axes can be traced by the end-effector pointer.

The EE_frame coordinate system is defined in robot coordinates by its origin and its axes (which are parallel to the robot coordinate frame) by the homogenous transformation ${}^{R}T_{EE}$ as

$$ {}^{R}T_{EE} = \begin{bmatrix} 1 & 0 & 0 & -2.33 \\ 0 & 1 & 0 & 570.00 \\ 0 & 0 & 1 & 214.00 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad (C.1) $$

The position vector in this transformation is in millimeters. The EE_frame coordinate system is defined in RDS coordinates by calculating range data for points along the EE_frame x_{EE}, y_{EE}, and z_{EE} axes as pointed to by the calibration pointer. The points along the axes must permit the laser stripe to intersect the calibration pointer tip, thus, points are

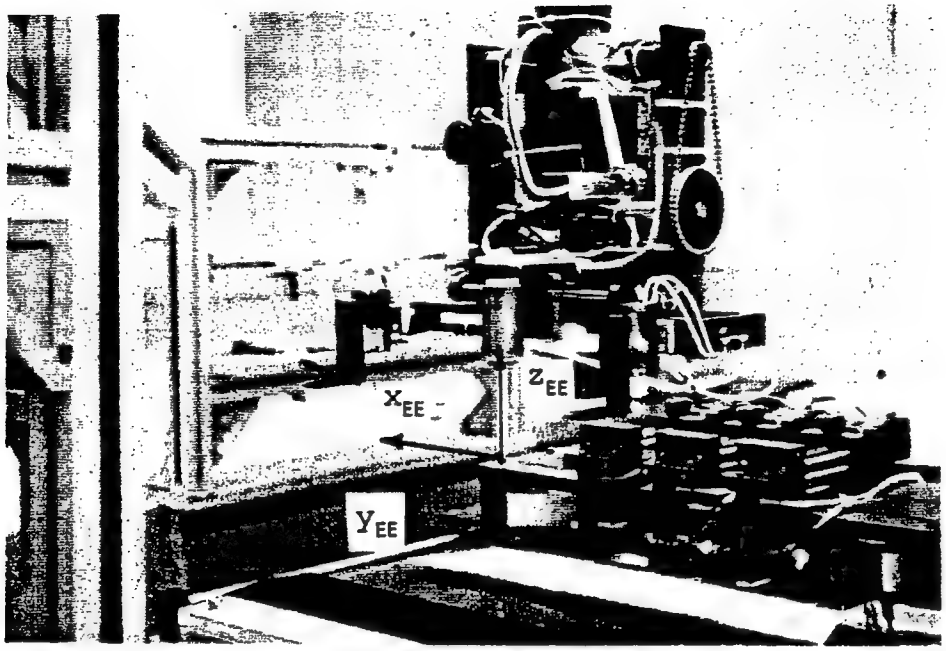


Figure C.1. The intermediate frame EE_frame.

determined by first projecting the laser stripe and then jogging the robot along the selected EE_frame axis to meet the stripe.

The EE_frame origin is determined with one point, and the EE_frame coordinate axes are defined as directions in the RDS coordinate system. An axis direction is determined by comparing the range data for points along the axis. For instance, the x_{RDS}, y_{RDS}, and z_{RDS} components of the vector difference between two points on an EE_frame axis are normalized to give a unit vector in the direction of this axis with respect to RDS coordinates. Since this unit vector is an approximation involving RDS uncertainty, several points along an EE_frame axis are used to obtain unit vectors representing the axis direction. The resulting unit vector directions are averaged to yield a "best fit" which minimizes error in the axis direction. The EE_frame axis directions as determined in RDS coordinates compose the rotation matrix ${}^{RDS}R_{EE}$ as

$$ {}^{RDS}R_{EE} = \begin{bmatrix} -0.9287 & 0.3096 & 0.2502 \\ 0.0079 & 0.6301 & -0.7787 \\ -0.3707 & -0.7121 & -0.5754 \end{bmatrix}. \qquad (C.2) $$

The determinant of matrix (C.2) is 0.9995. Since the determinant of a rotation matrix should be unity to satisfy the orthonormal condition, the directions of the individual EE_frame axes are considered approximately orthonormal.

The transformation between the EE_frame and RDS coordinate systems is given by

$$ {}^{RDS}T_{EE} = \begin{bmatrix} -0.9287 & 0.3096 & 0.2502 & -13.62 \\ 0.0079 & 0.6301 & -0.7787 & -71.64 \\ -0.3707 & -0.7121 & -0.5754 & 1118.217 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad (C.3) $$

By using (C.1) and (C.3), the following matrix algebra relates RDS coordinates to robot coordinates:

$$ {}^{EE}T_{RDS} = [{}^{RDS}T_{EE}]^{-1}, \quad \text{and} \qquad (C.4) $$

$$ {}^{R}T_{RDS} = {}^{R}T_{EE} * {}^{EE}T_{RDS}. \qquad (C.5) $$

This completes the derivation for the desired transformation ${}^{R}T_{RDS}$ which relates range data to robot coordinates by using

$$ V_R = {}^{R}T_{RDS} * V_{RDS}. \qquad (C.6) $$

Vector V_R and V_{RDS} are position vectors for a point with reference to the robot and RDS coordinate frames, respectively.

Appendix D

System Operation

Three programs are executed simultaneously to control the AAW workstation through a collar turning and pressing cycle. The programs are named SS??.exe, VC??.exe, and RC??.v2 to correspond with the respective host controller. The wildcard symbol ?? is a two digit identification representing a matched set of three programs, designed to be operated together by the respective controllers. Three sets of programs are available for system operation: SS38.exe, VC38.exe, and RC38.v2; SS40.exe, VC40.exe, and RC40.v2; and SS50.exe, VC50.exe, and RC50.v2.

The 38-program set is a complete operational program involving operator prompts throughout the system procedure. The prompts are an aid in training an operator as well as a troubleshooting tool. The 40-program set is the complete operational program without prompts. The 50-program set is a reduced version of the 38-program set whose cycle operation includes through only the collar loading function. The 50-program set is given in Appendix E with accompanying header files.

A sequence of steps follows for operating the system with the 50-program set. Each of the other sets is executed similarly.

1. Power up each of the PC controllers and the robot controller. Bring up the MSC\EXE subdirectories for each of the computers.

2. Execute the program AAWOFF.exe on the SS controller after the power to the peripherals has been switched on. This will initiate the I/O output signals. Push the green Program Start button on the robot controller to start the robot initialization, which takes approximately three minutes. Pressure to the robot must be 80 psi minimum.

3. Load RC50.v2 into the robot controller buffer from the c: drive. Execute the initialization program AAW.exe on the SS controller. This is a routine to home the motors in the workstation. The robot speed can be set with the command SPEED 45 for brisk action.

4. The system is now ready for operation. Enter the command EX RC50 in the robot computer, VC50 in the VC, and SS50 in the SS, and follow the prompts.

5. A selection is available between manually and automatically loading the collar on the end-effector. Choose one or the other, and continue the program. The automatic selection will use the destacker device for collar retrieval. Manual loading requires placing the unlined collar ply between the grippers with the fabric tensioned to prevent the ply from sagging and interfering with the bridge of the turning device. The collar should be centered side to side between the grippers, and the edge of the unlined ply should be 1/4" distant from the back edge of the gripper pads. Manual collar loading requires a person to load and an operator.

6. Once the robot presents the collar to the pressing station, the operator is prompted for two threshold values, one for each the left and right collar point regions. Values of 180 and 160 have been determined experimentally to yield isolated stripes without excessive noise.

7. The process continues until the collar is loaded on the vacuum surface. Another cycle is possible with the same commands: EX RC50, VC50, and SS50.

Appendix E

Source Code Listing

Vision Computer (VC) Source Code

```
/*********************************/
/* VC50.C                        */
/* AAW vision controller code */
/*                               */
/*********************************/

#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <strstuff.h>
#include <vccom2.h>
#include <vision.h>
#include <moses.h>
#include <kishore.h>

ts_main()
{
 struct char_pckt   ch_msg[NUMPCKTS];
 struct packet      msg[NUMPCKTS];
 struct packet_data p_data[NUMPCKTS];
 int                numpckts, charpos;
 char               type[6];
 int                nl, nr, ind, indl=1, indr=1, flag;
 int                ind171, ind173, ind175;
 int                ind224, ind226, ind228;
 struct stripe      line[8];
 struct R_xyz       colpoint_left, px_left, colpoint_right, px_right;
 struct R_xyz       delta, col_frame;
 int                row, column, point_array[10], coeff_array[9], i;
 FILE               *fpoint1, *fpoint2;

 xmitp_ss = fopen("xmitss.fil", "w+");
 recvp_ss = fopen("recvss.fil", "w+");
 com_setup_vc(port_ss, BUFFSIZE,
                BAUD19200, EVENPAR, DATA8, STOP1, CTSREQD);
 msgptr = spawn("Mssge_Handler_VC", 0x1200, mssge_handler_vc);
 signal_ss();
 recv_ss(ch_msg);
/*********************
 *             *
 * START OF PLAN.  *
 *             *
 *********************/
 /*********************************/
 /* initialize vision software */
 /*********************************/
 is_initialize();  /* initialize vision software */
 is_set_sync_source(EXT_SYNC);
 clear();
 is_select_channel(1);
 select(0);
 is_set_foreground(200);
 is_display(1);
 is_passthru();
 /**********************/
```

```
/* open output files */
/**********************/
fpoint1 = fopen("vc50a.out", "w");
fpoint2 = fopen("vc50b.out", "w");
signal_ss();          /* vc 05 */
ilut( 20, 3);
for(i=0; i<9; coeff_array[i++] = 1);
sleep(3);
is_select_channel(2);
signal_ss();          /* vc 06 */
sleep(5);
is_select_channel(1);
recv_mssge_ss(type, p_data);
numpckts = 0;
charpos  = 0;
numpckts = dis_int_data(&nl, p_data, &charpos, numpckts);
numpckts = dis_int_data(&nr, p_data, &charpos, numpckts);
is_freeze_frame();
is_select_output_frame(4);
/***************************/
/* acquire image of collar */
/***************************/
is_acquire( 4, 1);
signal_ss();                    /* vc 07 */
/***********************************/
/* assign binary threshold values */
/***********************************/
ilut( nl, 4);
is_select_ilut(4);
ilut( nr, 5);
/*****************************************/
/* compose RDS to Robot transformation */
/*****************************************/
ind = compose_TRDStoR();
if(ind == 0)
{
  printf("Function compose_TRDStoR() has failed!\n");
}
printf("The thresholds selected are:  %d  %d\n", nl, nr);
signal_ss();          /* vc 10 */
is_select_output_frame(1);
/*****************************************/
/* acquire image of stripe 0 and filter */
/*****************************************/
is_acquire( 1, 1);  /* take picture with init lefthand stripe */
is_convolve( 1, 13, 3, 3, coeff_array, 1);
is_do_alu( 13, 3, 8, 0, 16, 1);
/*************************************************************/
/* initiate linescan and edgefind bug for first stripe */
/*************************************************************/
ind171 = define_stripe( &line[0], 1, 'l', 175, 171, fpoint1);
if(ind171 != 0)
{
  printf("Line 171 has problems!\n");
}
signal_ss();                    /* vc 20 */
/*************************************************/
/* signal for new mirror angle to give stripe 1 */
/*************************************************/
signal_ss();          /* vc 30 */
is_select_output_frame(2);
/*****************************************/
/* acquire stripe 1 and image process */
/*****************************************/
is_acquire( 2, 1);
```

```
is_convolve( 2, 13, 3, 3, coeff_array, 1);
is_do_alu( 13, 3, 8, 0, 16, 2);
ind173 = define_stripe( &line[1], 2, 'l', 160, 173, fpoint1);
if(ind173 != 0)
{
 printf("Line 173 has problems!\n");
}
is_or(1, 2, 3);
is_frame_copy(3, 1);
is_select_output_frame(3);
signal_ss();          /* vc 30b */
signal_ss();                    /* vc 40 */
/*****************************************/
/* acquire stripe 2 and image process */
/*****************************************/
is_acquire( 2, 1);
is_convolve( 2, 13, 3, 3, coeff_array, 1);
is_do_alu( 13, 3, 8, 0, 16, 2);
ind175 = define_stripe( &line[2], 2, 'l', 145, 175, fpoint1);
if(ind175 != 0)
{
 printf("Line 175 has problems!\n");
}
/*********************************************************/
/* if stripes 0,1,2 were successfully image processed, */
/* then determine collar point location                  */
/*********************************************************/
else if((ind171 == 0) && (ind173 == 0))
{
 ind! = collar_point( line[2], line[1], line[0],
                &colpoint_left, &px_left, fpoint1);
 if(ind! != 0)
 {
  printf("The left collar_point routine failed!!");
 }
 else
 {
  printf("The Robot coords for the left collar point are:\n");
  printf("        %8.3lf %8.3lf %8.3lf\n\n",
  colpoint_left.x, colpoint_left.y, colpoint_left.z);
 }
}
is_or(1, 2, 3);
is_frame_copy(3, 1);
is_select_output_frame(3);
signal_ss();          /* vc 40b */
/*******************************************/
/* display point location on video monitor */
/*******************************************/
if(ind! == 0)
{
 row    = (int) px_left.y + 5;
 column = (int) px_left.x;
 printf("The left collar point pixel values are: %d %d\n\n",
                                                    row, column);
 is_set_graphic_position( row, column);
 point_array[ 0] = (int) px_left.y;
 point_array[ 1] = column;
 is_draw_arc( 3, (int) px_left.y, (int) px_left.x, 360);
 is_frame_copy(3, 1);
 is_select_output_frame(3);
}
is_select_ilut(5);
signal_ss();                    /* vc 50 */
/*****************************************/
```

```
/* acquire stripe 3 and image process */
/*************************************/
is_acquire( 2, 1);
is_convolve( 2, 13, 3, 3, coeff_array, 1);
is_do_alu( 13, 3, 8, 0, 16, 2);
ind224 = define_stripe( &line[3], 2, 'r', 150, 224, fpoint1);
if(ind224 != 0)
{
  printf("Line 224 has problems!\n");
}
is_or(1, 2, 3);
is_frame_copy(3, 1);
is_select_output_frame(3);
signal_ss();          /* vc 50a */
signal_ss();                    /* vc 60 */
/*************************************/
/* acquire stripe 4 and image process */
/*************************************/
is_acquire( 2, 1);
is_convolve( 2, 13, 3, 3, coeff_array, 1);
is_do_alu( 13, 3, 8, 0, 16, 2);
ind226 = define_stripe( &line[4], 2, 'r', 160, 226, fpoint1);
if(ind226 != 0)
{
  printf("Line 226 has problems!\n");
}
is_or(1, 2, 3);
is_frame_copy(3, 1);
is_select_output_frame(3);
signal_ss();          /* vc 60a */
signal_ss();          /* vc 70 */
/*************************************/
/* acquire stripe 5 and image process */
/*************************************/
is_acquire( 2, 1);
is_convolve( 2, 13, 3, 3, coeff_array, 1);
is_do_alu( 13, 3, 8, 0, 16, 2);
ind228 = define_stripe( &line[5], 2, 'r', 175, 228, fpoint1);
/*************************************************************/
/* if stripes 3,4,5 were successfully image processed, */
/* then determine collar point location                  */
/*************************************************************/
if(ind228 != 0)
{
  printf("Line 228 has problems!\n");
}
else if (( ind224 == 0 ) && ( ind226 == 0))
{
  indr = collar_point( line[3], line[4], line[5],
                  &colpoint_right, &px_right, fpoint1);
  if(indr != 0)
  {
    printf("The right collar_point routine failed!\n");
  }
  else
  {
    printf("The Robot coords for the right collar point are:\n");
    printf("        %8.3lf %8.3lf %8.3lf\n\n",
    colpoint_right.x, colpoint_right.y, colpoint_right.z);
  }
}
is_or(1, 2, 3);
is_frame_copy(3, 1);
is_select_output_frame(3);
signal_ss();                    /* vc 70a */
```

```
/************************************/
/* display point location on monitor */
/************************************/
if(indr == 0)
{
  row    = (int) px_right.y + 5;
  column = (int) px_right.x;
  printf("The right collar point pixel values are: %d %d\n\n",
                                                    row, column);
  is_set_graphic_position( row, column);
  is_draw_arc( 3, (int) px_right.y, (int) px_right.x, 360);
  if(indl == 0)
  {
    is_set_graphic_position( (int) px_right.y, column);
    is_draw_lines( 3, 1, point_array);
  }
}
if((indl == 0) && (indr == 0))
{
  length = difference( colpoint_left.x, colpoint_left.y, colpoint_left.z,
          colpoint_right.x, colpoint_right.y, colpoint_right.z);
  /************************************************/
  /* determine collar_frame for collar position */
  /*      and delta changes to x-y plane          */
  /************************************************/
  collar_frame( colpoint_left, colpoint_right,
                                          &delta, &col_frame, fpoint1);
  printf("\nThe length between points is:  %8.3lf", length);
  printf("\nThe delta changes are: x: %5.2lf y: %5.2lf  theta: %5.2lf",
                                    delta.x, delta.y, delta.z);
  printf("\nThe collar frame has been located at: x: %5.2lf y: %5.2lf\n",
                                    col_frame.x, col_frame.y);
  fprintf(fpoint1,"\nThe length between points is:  %8.3lf", length);
  fprintf(fpoint1,"\nThe delta changes are: ");
  fprintf(fpoint1,"x: %5.2lf y: %5.2lf theta: %5.2lf",
                                    delta.x, delta.y, delta.z);
  fprintf(fpoint1,"\nThe collar frame has been located at: ");
  fprintf(fpoint1,"x: %5.2lf y: %5.2lf\n\n", col_frame.x, col_frame.y);
}
else
{
  flag = 0;
  delta.x = delta.y = delta.z = col_frame.x = col_frame.y = 0.0;
}
savefile( 3);
is_set_sync_source(INT_SYNC);
is_add(3, 4, 5);
is_select_output_frame(5);
savefile( 5);
/*******************************************/
/* send delta changes to System Supervisor */
/*******************************************/
numpckts = 0;
charpos  = 0;
numpckts = asm_double_data(delta.x, p_data, &charpos, numpckts, 1);
xmit_mssge_ss(numpckts, "ROBOT", p_data);
numpckts = 0;
charpos  = 0;
numpckts = asm_double_data(delta.y, p_data, &charpos, numpckts, 1);
xmit_mssge_ss(numpckts, "ROBOT", p_data);
numpckts = 0;
charpos  = 0;
numpckts = asm_double_data(delta.z, p_data, &charpos, numpckts, 1);
xmit_mssge_ss(numpckts, "ROBOT", p_data);
numpckts = 0;
```

```
 charpos  = 0;
 numpckts = asm_double_data(col_frame.x, p_data, &charpos, numpckts, 1);
 xmit_mssge_ss(numpckts, "ROBOT", p_data);
 numpckts = 0;
 charpos  = 0;
 numpckts = asm_double_data(col_frame.y, p_data, &charpos, numpckts, 1);
 xmit_mssge_ss(numpckts, "ROBOT", p_data);
 numpckts = 0;
 charpos  = 0;
 numpckts = asm_int_data(flag, p_data, &charpos, numpckts, 1);
 xmit_mssge_ss(numpckts, "ROBOT", p_data);
 is_set_sync_source(EXT_SYNC);
 select(0);
 is_select_ilut(0);
 is_passthru();
 signal_ss();        /* vc 80 */
 fclose( fpoint1);
 fclose( fpoint2);
 clear();
 is_passthru();
/*****************
 *             *
 *  END OF PLAN.  *
 *             *
 *****************/
 printf("\nProgram Completed!\n");
 while(1)
  ;
}

/*************************************************************/
/*                  IMAGEPRO.H                               */
/*                                                           */
/*                  a header file to perform computations    */
/*                  with stripes, create a collar model,     */
/*                  and perform coordinate transformations   */
/*                                                           */
/*                  Spring 1991.  David R. Cultice           */
/*                                                           */
/*************************************************************/

#include <matstuff.h>
/*************************************************************/
/* define parameters for RDS and workspace transformations */
/*************************************************************/
#define FOCALPTX      17.00    /* focal pt for x direction, STO16 */
#define FOCALPTY      16.75    /* focal pt for y direction */
#define DISTANCED     17.0788  /* distance from focal pt to mirror, STO05 */
#define GAMMANGLE     64.0273  /* gamma angle, STO04 */
#define LAMBDANGLE    2.4727   /* lambda angle, STO15 */
#define PI            3.14159265
#define PEETORX            -2.330     /* x comp. of pos. vec. for TEEtoR */
#define PEETORY       570.000         /* y comp. of pos. vec. for TEEtoR */
#define PEETORZ       214.000         /* z comp. of pos. vec. for TEEtoR */
#define GRIP_LEFT_X   160.712
#define GRIP_LEFT_Y   647.100
#define GRIP_LEFT_Z   330.025
#define GRIP_RIGHT_X -160.636
#define GRIP_RIGHT_Y  647.100
#define GRIP_RIGHT_Z  330.025
#define TARGET_X           -2.33
#define TARGET_Y      689.202
#define TARGET_THETA       0.1286 /* degrees */

int bugmove( int, int *, int *);
```

```
void minmax( int, int, int *, int *, int *, int *, int [], int []);
int scan_right( int, int *, FILE *);
int scan_left( int, int *, FILE *);
int bug0( int, int, int, int *, int *, int *, int *, int [], int []);
int bug1( int, int, int, int *, int *, int *, int *, int [], int []);
int bug2( int, int, int, int *, int *, int *, int *, int [], int []);
int bug3( int, int, int, int *, int *, int *, int *, int [], int []);
void rds( double, double, double, double, double, double, FILE *);
double difference( double, double, double, double, double, double);
void triangulate( double, double, double,
                  double *, double *, double *, FILE *);
int compose_TRDStoR();
int rdstrans( double, double, double, double [], FILE *);
double intersection( double, double, double, double);
struct stripe curve_fit( double, double, double, double, double, double);
void print_stripe( struct stripe);
double sqr( double);
struct    endpoint  {
          double    x, y;
};
struct    stripe    {
          struct  endpoint  endp1, endp2;
          double    slope, y_intercept;
          double    step_val;
};
struct    R_xyz     {
          double    x, y, z;
};
double                          TRDStoR[ 16];
/*************************************************************/
/* function called by VC to search for stripe and endpoints */
/*************************************************************/
int define_stripe( struct  stripe *line, int f_no, int leftright,
                   int yinit, int step, FILE *fp)
{
 int                ybug, xbug, xinit, i, ind;
 int                xmaxsum, xminsum;
 int                bugcase, n;
 int                ymax = 0, ymin = 512, n1, n2, xmax[ 10], xmin[ 10];
 double     xminave, xmaxave;

 if( leftright == 'r')
  xinit = scan_left( f_no, &yinit, fp);  /* scans from right side */
 else
  xinit = scan_right( f_no, &yinit, fp);
 printf("\nThe bug has been started at: %d ", xinit);
 if((0 <= xinit) && (xinit < 512))
 {
  ybug = yinit;
  xbug = xinit;
  if( leftright == 'r')
   bugcase = bug3( f_no, ybug, xbug, &ymax, &ymin, &n1, &n2, xmax, xmin);
  else
   bugcase = bug1( f_no, ybug, xbug, &ymax, &ymin, &n1, &n2, xmax, xmin);
  ind = bugmove( bugcase, &ybug, &xbug);
  while((!((xbug==xinit) && (ybug==yinit))) && (ind == 0))
  {
   switch( bugcase)
   {
        case 0:
         bugcase = bug0( f_no, ybug, xbug, &ymax, &ymin, &n1, &n2, xmax, xmin);
         ind = bugmove( bugcase, &ybug, &xbug);
         break;
        case 1:
         bugcase = bug1( f_no, ybug, xbug, &ymax, &ymin, &n1, &n2, xmax, xmin);
```

```
            ind = bugmove( bugcase, &ybug, &xbug);
            break;
          case 2:
            bugcase = bug2( f_no, ybug, xbug, &ymax, &ymin, &n1, &n2, xmax, xmin);
            ind = bugmove( bugcase, &ybug, &xbug);
            break;
          case 3:
            bugcase = bug3( f_no, ybug, xbug, &ymax, &ymin, &n1, &n2, xmax, xmin);
            ind = bugmove( bugcase, &ybug, &xbug);
            break;
        }
      }
      if( ind == 0)
        {
        xminsum = 0;
        xmaxsum = 0;
        for( i=0; i<n1; i++)
              xmaxsum += xmax[ i];
        xmaxave = (double) xmaxsum / (double) i;
        for( i=0; i<n2; i++)
              xminsum += xmin[ i];
        xminave = (double) xminsum / (double) i;
        fprintf( fp, "\nymax = %d, xmax = %6.2lf", ymax, xmaxave);
        fprintf( fp, "\nymin = %d, xmin = %6.2lf", ymin, xminave);
        (*line).endp1.x = xminave;
        (*line).endp1.y = (double) ymin;
        (*line).endp2.x = xmaxave;
        (*line).endp2.y = (double) ymax;
        if( xminave == xmaxave)
              xmaxave += .1;
        (*line).slope = ((double) ( ymin - ymax)) / ( xminave - xmaxave);
        (*line).y_intercept = (double) ymin - ((*line).slope * xminave);
        (*line).step_val = (double) step;
        return 0;
      }
      else
      {
        fprintf( fp, "\nThe bug has traveled outside its limits");
        return 1;
      }
    }
    else
    {
      fprintf( fp, "\nThe line scan has failed: NO THRESHOLDED LINE FOUND\n");
      return 1;
    }
}
/*************************************************************/
/* linescan function to scan for stripe in right direction */
/*************************************************************/
int scan_right( int f_no, int *vert, FILE *fp)
{
  int  xbug, dest_array[1];

  for( xbug=0; xbug<140; xbug++)
  {
    is_get_pixel( f_no, *vert, xbug, 1, dest_array);
    if( dest_array[0] == 255)
    {
      printf("\nlinescan 1");
      printf("\nlinescan 1");
      fprintf( fp, "\nThe line was found at: x = %d", xbug);
      fprintf( fp, "\nwith the first line scan");
      printf("\nend linescan 1");
      printf("\nend linescan 1");
```

```
    return ( xbug-1);
   }
  }
  for( xbug=0; xbug<140; xbug++)
  {
   is_get_pixel( f_no, (*vert+15), xbug, 1, dest_array);
   if( dest_array[0] == 255)
   {
    printf("\nlinescan 2");
    printf("\nlinescan 2");
    fprintf( fp, "\nThe line was found at: x = %d", xbug);
    fprintf( fp, "\nwith the second line scan");
    *vert = *vert + 15;
    printf("\nend linescan 2");
    printf("\nend linescan 2");
    return ( xbug-1);
   }
  }
  for( xbug=0; xbug<140; xbug++)
  {
   is_get_pixel( f_no, (*vert-15), xbug, 1, dest_array);
   if( dest_array[0] == 255)
   {
    printf("\nlinescan 3");
    printf("\nlinescan 3");
    fprintf( fp, "\nThe line was found at: x = %d", xbug);
    fprintf( fp, "\nwith the third line scan");
    *vert = *vert - 15;
    printf("\nend linescan 3");
    printf("\nend linescan 3");
    return ( xbug-1);
   }
  }
  return 512;
}
/*************************************************************/
/* linescan function to scan for stripe in left direction  */
/*************************************************************/
int scan_left( int f_no, int *vert, FILE *fp)
{
 int  xbug, dest_array[1];

 for( xbug=511; xbug>390; xbug--)
 {
  is_get_pixel( f_no, *vert, xbug, 1, dest_array);
  if( dest_array[0] == 255)
  {
   printf("\nlinescan 1");
   printf("\nlinescan 1");
   fprintf( fp, "\nThe line was found at: x = %d", xbug);
   fprintf( fp, "\nwith the first line scan");
   printf("\nend linescan 1");
   printf("\nend linescan 1");
   return ( xbug+1);
  }
 }
 for( xbug=511; xbug>390; xbug--)
 {
  is_get_pixel( f_no, (*vert+15), xbug, 1, dest_array);
  if( dest_array[0] == 255)
  {
   printf("\nlinescan 2");
   printf("\nlinescan 2");
   fprintf( fp, "\nThe line was found at: x = %d", xbug);
   fprintf( fp, "\nwith the second line scan");
```

```
      *vert = *vert + 15;
      printf("\nend linescan 2");
      printf("\nend linescan 2");
      return ( xbug+1);
    }
  }
  for( xbug=511; xbug>390; xbug--)
  {
    is_get_pixel( f_no, (*vert-15), xbug, 1, dest_array);
    if( dest_array[0] == 255)
    {
      printf("\nlinescan 3");
      printf("\nlinescan 3");
      fprintf( fp, "\nThe line was found at: x = %d", xbug);
      fprintf( fp, "\nwith the third line scan");
      *vert = *vert - 15;
      printf("\nend linescan 3");
      printf("\nend linescan 3");
      return ( xbug+1);
    }
  }
  return 512;
}
/*****************************************************/
/* four functions representing four directions       */
/*  to search about a pixel for a clear path         */
/*****************************************************/
int bug0( int f_no, int vert, int horiz, int *ymaxp, int *yminp, int *n1p,
        int *n2p, int xmax[], int xmin[])
{
  int dest_array[1];

  is_get_pixel( f_no, vert+1, horiz, 1, dest_array);
  if( dest_array[0] == 255)
    minmax( vert+1, horiz, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 3;
  is_get_pixel( f_no, vert, horiz+1, 1, dest_array);
  if( dest_array[0] == 255)
    minmax( vert, horiz+1, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 0;
  is_get_pixel( f_no, vert-1, horiz, 1, dest_array);
  if( dest_array[0] == 255)
    minmax( vert-1, horiz, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 1;
  is_get_pixel( f_no, vert, horiz-1, 1, dest_array);
  if( dest_array[0] == 255)
    minmax( vert, horiz-1, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 2;
}

int bug1( int f_no, int vert, int horiz, int *ymaxp, int *yminp, int *n1p,
        int *n2p, int xmax[], int xmin[])
{
  int dest_array[1];

  is_get_pixel( f_no, vert, horiz+1, 1, dest_array);
  if( dest_array[0] == 255)
    minmax( vert, horiz+1, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 0;
  is_get_pixel( f_no, vert-1, horiz, 1, dest_array);
```

```
  if( dest_array[0] == 255)
    minmax( vert-1, horiz, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 1;
  is_get_pixel( f_no, vert, horiz-1, 1, dest_array);
  if( dest_array[0] == 255)
    minmax( vert, horiz-1, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 2;
  is_get_pixel( f_no, vert+1, horiz, 1, dest_array);
  if( dest_array[0] == 255)
    minmax( vert+1, horiz, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 3;
}

int bug2( int f_no, int vert, int horiz, int *ymaxp, int *yminp, int *n1p,
        int *n2p, int xmax[], int xmin[])
{
  int dest_array[1];

  is_get_pixel( f_no, vert-1, horiz, 1, dest_array);
  if( dest_array[0] == 255)
    minmax( vert-1, horiz, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 1;
  is_get_pixel( f_no, vert, horiz-1, 1, dest_array);
  if( dest_array[0] == 255)
    minmax( vert, horiz-1, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 2;
  is_get_pixel( f_no, vert+1, horiz, 1, dest_array);
  if( dest_array[0] == 255)
    minmax( vert+1, horiz, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 3;
  is_get_pixel( f_no, vert, horiz+1, 1, dest_array);
  if( dest_array[0] == 255)
    minmax( vert, horiz+1, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 0;
}

int bug3( int f_no, int vert, int horiz, int *ymaxp, int *yminp, int *n1p,
        int *n2p, int xmax[], int xmin[])
{
  int dest_array[1];

  is_get_pixel( f_no, vert, horiz-1, 1, dest_array);
  if( dest_array[0] == 255)
    minmax( vert, horiz-1, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 2;
  is_get_pixel( f_no, vert+1, horiz, 1, dest_array);
  if( dest_array[0] == 255)
    minmax( vert+1, horiz, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 3;
  is_get_pixel( f_no, vert, horiz+1, 1, dest_array);
  if( dest_array[0] == 255)
    minmax( vert, horiz+1, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 0;
  is_get_pixel( f_no, vert-1, horiz, 1, dest_array);
  if( dest_array[0] == 255)
```

```
    minmax( vert-1, horiz, ymaxp, yminp, n1p, n2p, xmax, xmin);
  else
    return 1;
}
/*****************************************************************/
/* function to move bug one pixel in unblockedopen direction */
/*****************************************************************/
int bugmove( int bugcase, int *ybugp, int *xbugp)
{
  switch( bugcase)
  {
    case 0:
      *xbugp += 1;
      break;
    case 1:
      *ybugp -= 1;
      break;
    case 2:
      *xbugp -= 1;
      break;
    case 3:
      *ybugp += 1;
      break;
  }
  if((*xbugp < 0) || (*xbugp >= 512))
    return 1;
  if((*ybugp < 0) || (*ybugp >= 512))
    return 1;
  return 0;
}
/*****************************************/
/* function holds the minimum and maximum */
/* position of the bug in the y direction */
/* to define the stripe enpoints          */
/*****************************************/
void minmax( int vert, int horiz, int *ymaxp, int *yminp, int *n1p, int *n2p,
             int xmax[], int xmin[])
{
  static int yprev, xprev;

  if((vert != yprev) || (horiz != xprev))
  {
    yprev = vert;
    xprev = horiz;
    if( vert > *ymaxp)
    {
      *ymaxp = vert;
      *n1p = 0;
    }
    if( vert == *ymaxp)
    {
      xmax[ *n1p] = horiz;
      *n1p += 1;
    }
    if( vert < *yminp)
    {
      *yminp = vert;
      *n2p = 0;
    }
    if( vert == *yminp)
    {
      xmin[ *n2p] = horiz;
      *n2p += 1;
    }
  }
```

```
}

void rds( double ypixel1, double xpixel1, double step1,
    double ypixel2, double xpixel2, double step2, FILE *fpoint)
{
 double     RDSx1, RDSy1, RDSz1, RDSx2, RDSy2, RDSz2;
 double     VectorR1[4], VectorR2[4];
 double     xave, yave, zave, length;
 int          ind1, ind2;

 triangulate( ypixel1, xpixel1, step1, &RDSx1, &RDSy1, &RDSz1, fpoint);
 ind1 = rdstrans( RDSx1, RDSy1, RDSz1, VectorR1, fpoint);
 triangulate( ypixel2, xpixel2, step2, &RDSx2, &RDSy2, &RDSz2, fpoint);
 ind2 = rdstrans( RDSx2, RDSy2, RDSz2, VectorR2, fpoint);
 xave = (VectorR1[0] + VectorR2[0])/2;
 yave = (VectorR1[1] + VectorR2[1])/2;
 zave = (VectorR1[2] + VectorR2[2])/2;
 length = difference( RDSx1, RDSy1, RDSz1, RDSx2, RDSy2, RDSz2);
 fprintf( fpoint, "\nThe 3-D coords for the stripe center are: \n");
 fprintf( fpoint, "      x: %8.3lf  y: %8.3lf  z: %8.3lf\n",
                                  xave, yave, zave);
 fprintf( fpoint, "The length is %8.3lf mm\n", length);
}

double difference( double RDSx1, double RDSy1, double RDSz1,
                   double RDSx2, double RDSy2, double RDSz2)
{
 double diff, delx, dely, delz;

 delx = RDSx1 - RDSx2;
 dely = RDSy1 - RDSy2;
 delz = RDSz1 - RDSz2;
 diff = sqrt(delx*delx + dely*dely + delz*delz);
 return diff;
}
/*************************************************/
/* triangulate for RDS coordinates: x,y,z.  */
/*************************************************/
void triangulate( double ypixel, double xpixel, double step,
                  double *RDSx, double *RDSy, double *RDSz, FILE *fpoint)
{
 double STO12, STO02, STO11, STO03, STO06;
 int        ind;

 fprintf( fpoint, "\npixel set:      x:%10.3lf  y:%10.3lf  step:%10.3lf",
     xpixel, ypixel, step);

 STO12 = (255.0 - xpixel)*8.8/(510.0*FOCALPTX);
 STO02 = atan( STO12);
 STO11 = (245.0 - ypixel)*6.6/(492.0*FOCALPTY);
 STO03 = (( step * 0.45) + LAMBDANGLE) * PI/180.0;
 STO06 = STO02 + (GAMMANGLE * PI/180.0);
 *RDSz = cos(STO02) * tan(STO03) * DISTANCED * 25.4
           /  ((tan(STO06) + tan(STO03)) * cos(STO06));
 *RDSx = -STO12 * *RDSz;
 *RDSy = -STO11 * *RDSz;

 fprintf( fpoint, "\nRDS coords:  x:%10.3lf  y:%10.3lf  z:%10.3lf",
                                  *RDSx, *RDSy, *RDSz);
}
/*****************************************************/
/* transform rds coordinates to robot coordinates */
/*****************************************************/
int rdstrans( double RDSx, double RDSy, double RDSz, double VectorR[],
              FILE *fpoint)
```

```
{
 double     VectorRDS[ 4];
 int          ind;

 /* compile Vectors */
 VectorRDS[ 0] = RDSx;
 VectorRDS[ 1] = RDSy;
 VectorRDS[ 2] = RDSz;
 VectorRDS[ 3] = 1.0;

 ind = mulmxvc( TRDStoR, 4, 4, VectorRDS, 4, VectorR);
 if( ind == 0)
 {
  printf("\nThe mult of matrix and vector is not possible!!");
  printf("\nSystem aborted!!");
  return 0;
 }
 fprintf(fpoint, "\nrobot coords:    x:%10.3lf  y:%10.3lf  z:%10.3lf\n",
         VectorR[0], VectorR[1], VectorR[2]);
 return 1;
}
/*************************************************/
/* derive location for collar point in robot   */
/* coordinates with 3 stripe endpoint pairs    */
/* and 3 corresponding mirror angles           */
/*************************************************/
int collar_point( struct stripe major, struct stripe middle,
                  struct stripe minor, struct R_xyz *colpoint,
                  struct R_xyz *px, FILE *fp)
{
 struct  stripe  temp, top, bottom;
 double    x_int, y_int, ratio1, ratio2;
 double    x_point, y_point, distance;
 double    RDSx, RDSy, RDSz, VectorR[ 4];
 int               ind;

 temp.slope = -1.0 / major.slope;
 temp.y_intercept = middle.endp1.y - (temp.slope * middle.endp1.x);
 x_int = intersection( temp.slope, temp.y_intercept, major.slope,
                       major.y_intercept);
 y_int = (temp.slope * x_int) + temp.y_intercept;
 ratio1 = difference( x_int, y_int, 0.0, middle.endp1.x,
                      middle.endp1.y, 0.0);
 ratio1 = fabs( major.step_val - middle.step_val) / ratio1;
 temp.slope = -1.0 / middle.slope;
 temp.y_intercept = minor.endp1.y - (temp.slope * minor.endp1.x);
 x_int = intersection( temp.slope, temp.y_intercept, middle.slope,
                       middle.y_intercept);
 y_int = (temp.slope * x_int) + temp.y_intercept;
 ratio2 = difference( x_int, y_int, 0.0, minor.endp1.x,
                      minor.endp1.y, 0.0);
 ratio2 = fabs( middle.step_val - minor.step_val) / ratio2;
 ratio1 = (ratio1 + ratio2) / 2.0;
 top = curve_fit( major.endp1.x, major.endp1.y, middle.endp1.x,
                  middle.endp1.y, minor.endp1.x, minor.endp1.y);
 bottom = curve_fit( major.endp2.x, major.endp2.y, middle.endp2.x,
                     middle.endp2.y, minor.endp2.x, minor.endp2.y);
 x_point = intersection( top.slope, top.y_intercept, bottom.slope,
                         bottom.y_intercept);
 y_point = (top.slope * x_point) + top.y_intercept;
 (*px).x = x_point;
 (*px).y = y_point;
 temp.slope = -1.0 / major.slope;
 temp.y_intercept = y_point - (temp.slope * x_point);
 x_int = intersection( temp.slope, temp.y_intercept, major.slope,
```

```
                                    major.y_intercept);
  y_int = (temp.slope * x_int) + temp.y_intercept;
  distance = difference( x_int, y_int, 0.0, x_point, y_point, 0.0);
  if( major.endp1.x > x_point)
    temp.step_val = major.step_val - (distance * ratio1);
  else
    temp.step_val = major.step_val + (distance * ratio1);
  triangulate( y_point, x_point, temp.step_val, &RDSx, &RDSy, &RDSz, fp);
  ind = rdstrans( RDSx, RDSy, RDSz, VectorR, fp);
  if( ind == 0)
  {
    fprintf( fp, "\nrdstrans routine has not been carried out!!");
    return 1;
  }
  else
  {
    (*colpoint).x = VectorR[0];
    (*colpoint).y = VectorR[1];
    (*colpoint).z = VectorR[2];
    return 0;
  }
}
/*************************************/
/* assign coordinate frame to collar */
/*************************************/
void collar_frame( struct R_xyz colpoint_left, struct R_xyz colpoint_right,
                   struct R_xyz *delta, struct R_xyz *col_frame, FILE *fp)
{
  double  delta_x_left, delta_x_right, delta_y_left, delta_y_right;
  double  delta_z_left, delta_z_right;
  double  x_eff_left, x_eff_right, y_eff_left, y_eff_right;
  double  collar_x, collar_y, collar_theta;

  delta_x_left       = colpoint_left.x - GRIP_LEFT_X;
  delta_y_left       = colpoint_left.y - GRIP_LEFT_Y;
  delta_z_left       = colpoint_left.z - GRIP_LEFT_Z;
  delta_x_right = colpoint_right.x - GRIP_RIGHT_X;
  delta_y_right = colpoint_right.y - GRIP_RIGHT_Y;
  delta_z_right = colpoint_right.z - GRIP_RIGHT_Z;

  x_eff_left = GRIP_LEFT_X + delta_x_left;
  y_eff_left = GRIP_LEFT_Y + sqrt( sqr( delta_y_left) + sqr( delta_z_left));
  x_eff_right = GRIP_RIGHT_X + delta_x_right;
  y_eff_right = GRIP_RIGHT_Y + sqrt( sqr( delta_y_right) +
                                                        sqr( delta_z_right));
  collar_x = 0.5*( x_eff_left + x_eff_right);
  collar_y = 0.5*( y_eff_left + y_eff_right);
  collar_theta = atan2(( y_eff_left - y_eff_right),
                                        ( x_eff_left - x_eff_right))* 180/PI;
  printf("\nThe collar_frame attributes are:  %6.2lf %6.2lf %6.2lf",
                                    collar_x, collar_y, collar_theta);
  (*delta).x = TARGET_X - collar_x;
  (*delta).y = TARGET_Y - collar_y;
  (*delta).z = TARGET_THETA - collar_theta;
  (*col_frame).x = collar_x;
  (*col_frame).y = collar_y;
}

double sqr( double var)
{
  return var * var;
}
/*****************************************************/
/* function to determine intersection of two lines */
/*****************************************************/
```

```
double intersection( double slope1, double y_int1, double slope2,
                     double y_int2)
{
  double x_int;

  x_int = (y_int1 - y_int2) / (slope2 - slope1);
  return x_int;
}
/*************************************************/
/* fit least squares regression to endpoints */
/*************************************************/
struct stripe curve_fit( double x1, double y1, double x2, double y2,
                                    double x3, double y3 )
{
  struct stripe line;
  double  sigx, sigy, sigxy, sigxsigy, sig_x2, sigx_2;

  sigx = (x1 + x2 + x3);
  sigy = (y1 + y2 + y3);
  sigxy = x1 * y1 + x2 * y2 + x3 * y3;
  sigxsigy = sigx * sigy;
  sig_x2 = x1 * x1 + x2 * x2 + x3 * x3;
  sigx_2 = sigx * sigx;

  line.slope = (sigxy -  sigxsigy / 3.0) / ( sig_x2 -  sigx_2 / 3.0);
  line.y_intercept = sigy / 3.0 - line.slope * sigx / 3.0;

  return line;
}

void print_stripe( struct stripe line)
{
  printf("\nThe pixel coords for the top endpoint; x = %d y = %d",
          (int) line.endp1.x, (int) line.endp1.y);
  printf("\nThe pixel coords for the bottom endpoint; x = %d y = %d",
          (int) line.endp2.x, (int) line.endp2.y);
  printf("\nThe slope is %8.5lf ", line.slope);
  printf("\nThe y_intercept is %8.5lf ", line.y_intercept);
  printf("\nThe step value for this stripe is %d", (int) line.step_val);
}
/*****************************************************/
/* assemble RDS to Robot coordinate transformation */
/*****************************************************/
int compose_TRDStoR()
{
  double     TEEtoR[ 16], TEEtoRDS[ 16], TRDStoEE[ 16];
  int          i, ind, crow, ccol;

  /* create transformation matrices */
  for( i = 0; i < 16; TEEtoR[ i++] = 0.0);
  TEEtoR[ 0] =      1.0;
  TEEtoR[ 3] =      PEETORX;
  TEEtoR[ 5] =      1.0;
  TEEtoR[ 7] =      PEETORY;
  TEEtoR[10] =      1.0;
  TEEtoR[11] =      PEETORZ;
  TEEtoR[15] =      1.0;

  TEEtoRDS[ 0] =   -.9287;
  TEEtoRDS[ 1] =    .3096;
  TEEtoRDS[ 2] =    .2502;
  TEEtoRDS[ 3] = -13.62  ;
  TEEtoRDS[ 4] =    .0079;
  TEEtoRDS[ 5] =    .6301;
  TEEtoRDS[ 6] =   -.7787;
```

```
  TEEtoRDS[ 7] = -71.64  ;
  TEEtoRDS[ 8] =   -.3707;
  TEEtoRDS[ 9] =   -.7121;
  TEEtoRDS[10] =   -.5754;
  TEEtoRDS[11] =1118.217 ;
  TEEtoRDS[12] =   0.0   ;
  TEEtoRDS[13] =   0.0   ;
  TEEtoRDS[14] =   0.0   ;
  TEEtoRDS[15] =   1.0   ;

  /* invert TEEtoRDS to TRDStoEE Transformation matrix */
  inverse( TEEtoRDS, TRDStoEE);
  ind = mulmx( TEEtoR, 4, 4, TRDStoEE, 4, 4, TRDStoR, &crow, &ccol);
  if( ind == 0)
  {
    printf("\nThe mult of the two matrices was not possible!!");
    printf("\nSystem aborted!!");
    return 0;
  }
  return 1;
}

/*************************************/
/*  VISION.H                         */
/*                                   */
/*  header file for vision functions  */
/*************************************/

#include <isdefs.h>
#include <iserrs.h>

#define  EXT_SYNC     1
#define  INT_SYNC     0
#define  MAX_STR      80
#define  GREY_SCALE  256

void    pause();
void    sleep();
void    savefile();
void    select();
void    clear( void);
void    ilut( int, int);

/* wait for enter key */
void pause()
{
  while(keyin() != ENTRKEY)
    ;
}
/*******************************/
/* no operation for n seconds */
/*******************************/
void sleep(n)
int n;
{
  long timeval, time();

  timeval = time(0L);
  while(time(0L) < timeval + n);
}
/*************************************************/
/* save image file to operator selected filename */
/*************************************************/
void savefile( frame_no)
int frame_no;
```

```
{
 char file1[ MAX_STR];
 int spac_res_factor, ch;
 printf("\nDo you wish to save this frame to a file? (y or n): ");
 ch = getche();
 if ((ch == 'y') || (ch == 'Y'))
 {
  printf("\nEnter the path and filename.ext to store frame: ");
  scanf ( "%s", file1);
  printf("You entered %s.", file1);
  printf("\nEnter the spacial resolution factor (1 [full] - 8 incl.): ");
  scanf ( "%d", &spac_res_factor);
  is_save( frame_no, 1, spac_res_factor, 0, file1);  /*run-length encoded*/
  is_restore( frame_no, 0, 0, file1);
 }
}

void select( frame_no)
int frame_no;
{
 is_select_input_frame( frame_no);
 is_select_output_frame( frame_no);
}

/*********************************************/
/* assign binary threshold value for LUT    */
/*********************************************/
void ilut( int threshold, int ilut_number)
{
 int i, ilut_array[ GREY_SCALE];

 for( i=0; i<threshold; i++)
  ilut_array[i] = 0;
 for( i=threshold; i<256; i++)
  ilut_array[i] = 255;
 is_load_ilut( ilut_number, ilut_array);
}

void clear( void)
{
 int n;

 for( n=0; n<16; n++)
  is_frame_clear(n);
}

/*********************/
/*  MATRIX.H         */
/*  matrix functions */
/*********************/

#define  NDETSQR  36

int mulmx(a, arow, acol, b, brow, bcol, c, crowp, ccolp)
int    arow, acol, brow, bcol, *crowp, *ccolp;
double a[], b[], c[];
{
 int    i, j, k, ia, ib, ic;
 if(acol != brow)
  return 0;
 for(i = 1; i <= arow; i++)
  for(j = 1; j <= bcol; j++)
  {
   mxvc(&k, i, j, bcol);
   c[k] = 0.0;
```

```
    }
  for(i = 1; i <= arow; i++)
    for(j = 1; j <= bcol; j++)
      for(k = 1; k <= acol; k++)
      {
        mxvc(&ic, i, j, bcol);
        mxvc(&ia, i, k, acol);
        mxvc(&ib, k, j, bcol);
        c[ic] += a[ia]*b[ib];
      }
  *crowp = arow;
  *ccolp = bcol;
  return 1;
}

int inverse(a, ai, n)
double a[], ai[];
int    n;
{
  double b[NDETSQR], detm;
  int    nm, i, j, k, l, row, col, m, last, itrans;
  detm = det(a, n);
  if(detm == 0.0)
    return 0;
  nm = n - 1;
  last = n*n;
  for(i = 0; i < last; i++)
  {
    vcmx(i, &row, &col, n);
    m = 0;
    for(j = 1; j <= n; j++)
      for(k = 1; k <= n; k++)
      {
        if((j != row) && (k != col))
        {
          mxvc(&l, j, k, n);
          b[m] = a[l];
          m++;
        }
      }
    mxvc(&itrans, col, row, n);
    if((row + col) % 2)
      ai[itrans] = -det(b, nm)/detm;
    else
      ai[itrans] =  det(b, nm)/detm;
  }
  return 1;
}

int mulmxvc(a, arow, acol, b, brow, c)
double a[], b[], c[];
int    arow, acol, brow;
{
  int i, j, k;
  if(acol != brow)
    return 0;
  for(i = 0; i < arow; i++)
    c[i] = 0.0;
  for(i = 0; i < arow; i++)
    for(j = 0; j < brow; j++)
    {
      mxvc(&k, i+1, j+1, acol);
      c[i] += a[k]*b[j];
    }
  return 1;
```

```
}
```

System Supervisor (SS) Source Code

```
/*************************************/
/*  SS50.C                           */
/*                                   */
/* System Supervisor AAW Control Code */
/*                                   */
/*************************************/

#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <strstuff.h>
#include <handetc2.h>
#include <sscom2.h>

ts_main()
{
 char              resp[80], type[6];
 struct char_pckt   ch_msg[NUMPCKTS];
 struct packet      msg[NUMPCKTS];
 struct packet_data p_data[NUMPCKTS];
 int               numpckts, charpos;
 int               nl, nr, flag, dflag;
 double            fing_dist[3];
 struct R_xyz
 {
  double x, y, z;
 } delta, col_frame;
 system("cls");
 xmitp_vc = fopen("xmitvc.fil", "w+");
 recvp_vc = fopen("recvvc.fil", "w+");
 xmitp_rc = fopen("xmitrc.fil", "w+");
 recvp_rc = fopen("recvrc.fil", "w+");
 com_setup_ss(port_vc, BUFFSIZE,
              BAUD19200, EVENPAR, DATA8, STOP1, CTSREQD);
 com_setup_ss(port_rc, BUFFSIZE,
              BAUD19200, EVENPAR, DATA8, STOP1, NOCTS);
 al_initialize();
 al_select_board(1);
 al_reset();
 al_enable_for_output(0);
 al_enable_for_input(1);
 al_output_digital_value(0, 0x01, 0x00);  /* setting for RS-232 */
 kbdint = intatt(0x9, 0x100, kbd_isr, REPLACE, kbdptr);
 kbdptr = spawn("Kbd_Tsk", 0x400, kbd_tsk);
 msgptr = spawn("Mssge_Handler_SS", 0x1500, mssge_handler_ss);
 signal_vc();
 signal_rc();
 printf("\nPress <ENTER> to begin! ");
 fget_sline(stdin, resp);
/*********************
*                    *
* START OF PLAN.     *
*                    *
*********************/
 /**********************/
 /* initialize system */
 /**********************/
 init_aaw();
```

```
slideway(0);
grip(0);
pickers(0);
destack(1);
clamps(0);
blades(0);
vacuum_table_elevation(0);
trans_grippers_abs(0.0);
pitch_ee_fastabs(0.0);
pivot_abs(24.17);
rotate_mirror_abs(100L);
/***********************************/
/* code to interact with operator */
/***********************************/
signal_rc();          /* rc 001 */
printf("Load from destacker? (y/n)\n");
fget_sline(stdin, resp);
system("cls");
if(resp[0] == 'y')
  dflag = 1;
else
  dflag = 0;
numpckts = 0;
charpos  = 0;
numpckts = asm_int_data(dflag, p_data, &charpos, numpckts, 1);
xmit_mssge_rc(numpckts, "DELTA", p_data);
if(dflag)
{
  destack(0);
  signal_vc();        /* vc 05  */
  pickers(1);
  destack(1);
  signal_rc();        /* rc 001a */
  signal_rc();        /* rc 001b */
  grip(1);
  pickers(0);
  signal_rc();        /* rc 001c */
}
else
{
  printf("Load collar and press ENTER when done.");
  fget_sline(stdin, resp);
  system("cls");
  grip(1);
  printf("Press ENTER when operator is out of robot workspace.");
  fget_sline(stdin, resp);
  system("cls");
  signal_vc();          /* vc 05  */
  signal_rc();          /* rc 002 */
}
/***********************************************/
/* session to turn collar with turning device */
/***********************************************/
signal_rc();          /* rc 003 */
pivot_abs(0.0);
air_jets(1);
trap(1);
air_jets(0);
grip(0);
signal_rc();          /* rc 004 */
signal_rc();          /* rc 005 */
pivot_abs(85.0);
pitch_ee_fastabs(-12.8);
signal_rc();          /* rc 006 */
signal_rc();          /* rc 007 */
```

```
trans_grippers(0.5);
printf("Press ENTER to close grippers.");
fget_sline(stdin, resp);
system("cls");
grip(1);
trap(0);
signal_rc();        /* rc 008 */
signal_vc();        /* vc 06  */
pitch_ee_fastabs(-50.0);
trans_grippers(0.9375);
signal_rc();                    /* rc 009 */
/*****************************************/
/* beginning of laser striping session */
/*****************************************/
system("cls");
printf("Type in left and right binary threshold values to be used");
printf("\nfor the ILUTs (choose between 170 and 255):   ");
scanf("%d %d", &nl, &nr);
system("cls");
printf("You've selected values of:  %d  %d\n", nl, nr);
fget_sline(stdin, resp);
/*****************************************/
/* send chosen threshold values to VC */
/*****************************************/
numpckts = asm_int_data(nl, p_data, &charpos, numpckts, 0);
numpckts = asm_int_data(nr, p_data, &charpos, numpckts, 1);
xmit_mssge_vc(numpckts, "RDSSV", p_data);
signal_vc();                    /* vc 07 */
/* turn laser on */
laser(1);
/*****************************************/
/* rotate mirror to stripe position 0 */
/*****************************************/
rotate_mirror_abs(171L);  /* left-most laser line */
signal_vc();        /* vc 10 */
/* wait for VC to image process the stripe */
signal_vc();        /* vc 20 */
/*****************************************/
/* rotate mirror to stripe position 1 */
/*****************************************/
rotate_mirror(2L);
signal_vc();        /* vc 30 */
signal_vc();        /* vc 30b */
/*****************************************/
/* rotate mirror to stripe position 2 */
/*****************************************/
rotate_mirror(2L);
signal_vc();        /* vc 40 */
signal_vc();        /* vc 40b */
/*****************************************/
/* rotate mirror to stripe position 3 */
/*****************************************/
rotate_mirror_abs(224L);  /* direct laser line to right collar half */
signal_vc();        /* vc 50 */
signal_vc();        /* vc 50a */
rotate_mirror(2L);
signal_vc();        /* vc 60 */
signal_vc();        /* vc 60a */
ssleep();
rotate_mirror(2L);
swake();
signal_vc();        /* vc 70 */
signal_vc();        /* vc 70a */
/* turn off laser */
laser(0);
```

```
rotate_mirror_abs(100L);
vacuum_table_elevation(1);
while(!is_vacuum_table_up())
  printf("Waiting for vacuum table to go up.\n");
/*****************************************/
/* receive delta error vector            */
/* from VC and relay it to RC            */
/*****************************************/
recv_mssge_vc(type, p_data);
numpckts = 0;
charpos  = 0;
numpckts = dis_double_data(&delta.x, p_data, &charpos, numpckts);
recv_mssge_vc(type, p_data);
numpckts = 0;
charpos  = 0;
numpckts = dis_double_data(&delta.y, p_data, &charpos, numpckts);
recv_mssge_vc(type, p_data);
numpckts = 0;
charpos  = 0;
numpckts = dis_double_data(&delta.z, p_data, &charpos, numpckts);
recv_mssge_vc(type, p_data);
numpckts = 0;
charpos  = 0;
numpckts = dis_double_data(&col_frame.x, p_data, &charpos, numpckts);
recv_mssge_vc(type, p_data);
numpckts = 0;
charpos  = 0;
numpckts = dis_double_data(&col_frame.y, p_data, &charpos, numpckts);
recv_mssge_vc(type, p_data);
numpckts = 0;
charpos  = 0;
numpckts = dis_int_data(&flag, p_data, &charpos, numpckts);
numpckts = 0;
charpos  = 0;
numpckts = asm_double_data(delta.x, p_data, &charpos, numpckts, 1);
xmit_mssge_rc(numpckts, "DELTA", p_data);
numpckts = 0;
charpos  = 0;
numpckts = asm_double_data(delta.y, p_data, &charpos, numpckts, 1);
xmit_mssge_rc(numpckts, "DELTA", p_data);
numpckts = 0;
charpos  = 0;
numpckts = asm_double_data(delta.z, p_data, &charpos, numpckts, 1);
xmit_mssge_rc(numpckts, "DELTA", p_data);
numpckts = 0;
charpos  = 0;
numpckts = asm_double_data(col_frame.x, p_data, &charpos, numpckts, 1);
xmit_mssge_rc(numpckts, "DELTA", p_data);
numpckts = 0;
charpos  = 0;
numpckts = asm_double_data(col_frame.y, p_data, &charpos, numpckts, 1);
xmit_mssge_rc(numpckts, "DELTA", p_data);
numpckts = 0;
charpos  = 0;
numpckts = asm_int_data(flag, p_data, &charpos, numpckts, 1);
xmit_mssge_rc(numpckts, "DELTA", p_data);
/*****************************************************/
/* synchronize end-effector and vacuum with robot loading */
/*****************************************************/
pitch_ee_fastabs(-90.0);
signal_rc();                    /* rc 010 */
signal_rc();        /* rc 011 */
vacuum(1);
signal_rc();        /* rc 012 */
signal_rc();        /* rc 013 */
```

```
 signal_rc();                    /* rc 014 */
 signal_rc();                    /* rc 015 */
 signal_vc();                    /* vc 80 */
 /*****************/
 /* reset system */
 /*****************/
 vacuum(0);
 trans_grippers_abs(0.0);
 pitch_ee_fastabs(0.0);
 pivot_abs(24.17);
/*******************
 *               *
 * END OF PLAN. *
 *               *
 *******************/
 printf("\nProgram Completed!\n");
 while(1)
   ;
}


/*********************************************/
/* HANDETC.H                                 */
/*                                           */
/* utility file with DCX control commands */
/*********************************************/
#include <dcxcifc.h>
#include <atidefs.h>
#include <atlerrs.h>

void    init_rds();
void    laser();
void    rotate_mirror();
void    rotate_mirror_abs();
/***********************/
/* RDS initialization */
/***********************/
void init_rds()
{
 int            r;
 struct rpyfmt32 ch8, ch12;

 dcxcmd(0, 1, AXIS2 + AB, 0L);
 dcxcmd(0, 2, AXIS2 + SI, 1L, AXIS2 + SV, 2L);
 dcxcmd(0, 1, AXIS2 + SA, 3L);
 dcxcmd(0, 1, AXIS2 + MN, 0L);
 /* this initialized channel,as output,
    set output low: turned laser off   */
 dcxcmd(0, 1, TC, 12L);  /* This will verify that the flag is off sensor*/
 dcxrpy(0, sizeof(ch12), (int far *) &ch12);
 r = ch12.val;
 if(r)
 {
  dcxcmd(0, 2, AXIS2 + MR, -100L, AXIS2 + WS, 100L);
  dcxcmd(0, 1, NO);
 }
 r = 0;
 dcxcmd(0, 3, AXIS2 + DI, 0L, AXIS2 + VM, 0L, AXIS2 + GO, 0L);
 while(!r)
 {
  dcxcmd(0, 1, TC, 12L);
  dcxrpy(0, sizeof(ch12), (int far *) &ch12);
  r = ch12.val;  /* flag rotates until it trips sensor (ch12) */
  if(r)
    dcxcmd(0, 1, AXIS2 + ST, 0L);
```

```
 }
 wait_1s();
 dcxcmd(0, 1, AXIS2 + MN, 0L);
 while(1)   /* access step 0 output for motor 2: ch8*/
 {
   dcxcmd(0, 1, TC, 8L);
   dcxrpy(0, sizeof(ch8), (int far *) &ch8);
   r = ch8.val;
   if(r)
     dcxcmd(0, 2, AXIS2 + MR, -1L, AXIS2 + WS, 100L);
   else      /* motor 2 is at step 0 */
     break;
 }
 dcxcmd(0, 3, AXIS2 + SI, 2L, AXIS2 + SV, 10L, AXIS2 + SA, 6L);
 rotate_mirror(-186L);
 wait_100ms();
 dcxcmd(0, 2, AXIS2 + DH, 0L, AXIS2 + WS, 100L);
 rotate_mirror_abs(100L);
}
/*****************/
/* laser on/off  */
/*****************/
void laser(state)
int state;
{
 if(state)
   dcxcmd(0, 1, CN, 7L);
 else
   dcxcmd(0, 1, CF, 7L);
}
/*****************************************/
/* rotate mirror by 'step' relative steps */
/*****************************************/
void rotate_mirror(step)
long step;
{
 dcxcmd(0, 2, AXIS2 + MR, step, AXIS2 + WS, 255L);
 wait_100ms();
 dcxcmd(0, 1, NO);
}
/*****************************************/
/* rotate mirror by 'step' absolute steps */
/*****************************************/
void rotate_mirror_abs(step)
long step;
{
 dcxcmd(0, 2, AXIS2 + MA, step, AXIS2 + WS, 255L);
 wait_100ms();
 dcxcmd(0, 1, NO);
}
```

Robot Controller (RC) Source Code

```
.PROGRAM rc50()
;         for Adept MC1 Controller
;
          AUTO $mssge
          AUTO REAL deltax, deltay, theta, col_framex, col_framey
          AUTO REAL dflag, flag

          ATTACH (5)
          SPEED 50 ALWAYS
```

```
        ACCEL 120, 20
        signal_state = 0
        CALL signal_ss()
        SIGNAL -2001
        PCEXECUTE pc (), 0, 0
        WAIT SIG(2001)
        SIGNAL -2010
        CALL recv($mssge)
        REACT 1001, shutdown, 10
;*********************
;*                   *
;*  START OF PLAN.   *
;*                   *
;*********************
        TYPE "The program is running.."
        LEFTY
        SPEED 100
        MOVE loc[0]
        BREAK
        CALL signal_ss() ; rc 001
        CALL recv_mssge_ss($pckt.type, $pckt.data[])
        charpos = 1
        numpckts = 0
        CALL disassemb_int(dflag, $pckt.data[], charpos, numpckts)
        IF dflag THEN
           SPEED 150
           MOVE destack1
           BREAK
           CALL signal_ss() ; rc 001a
           SPEED 6
           MOVES destack2
           BREAK
           CALL signal_ss() ; rc 001b
           CALL signal_ss() ; rc 001c
           SPEED 25
           MOVE destack3
           BREAK
           SPEED 50
           MOVES destack1
           SPEED 120
           MOVES loc[2]
           BREAK
        ELSE
           CALL signal_ss() ; rc 002
           SPEED 150
           MOVES loc[1]
           SPEED 100
           MOVE loc[2]
           BREAK
        END
        MOVE loc[3]
        BREAK
        MOVE loc[4]
        BREAK
        CALL signal_ss() ; rc 003
        CALL signal_ss() ; rc 004
        MOVE loc[5]
        BREAK
        MOVE loc[6]
        BREAK
        CALL signal_ss() ; rc 005
        MOVE loc[7]
        BREAK
        CALL signal_ss() ; rc 006
        MOVE loc[8]
```

```
            BREAK
            SPEED 5
            MOVES loc[9]
            BREAK
            SPEED 10
            MOVES loc[10]
            BREAK
            CALL signal_ss() ; rc 007
            CALL signal_ss() ; rc 008
            SPEED 100
            MOVE loc[11]
            SPEED 240
            MOVE loc11a
            SPEED 200
            MOVE loc[12]
            BREAK
            CALL signal_ss() ; rc 009
;
; receive 'delta' error vector from SS
;
            CALL recv_mssge_ss($pckt.type, $pckt.data[])
            charpos = 1
            numpckts = 0
            CALL disassemb_real(deltax, $pckt.data[], charpos, numpckts)
            CALL recv_mssge_ss($pckt.type, $pckt.data[])
            charpos = 1
            numpckts = 0
            CALL disassemb_real(deltay, $pckt.data[], charpos, numpckts)
            CALL recv_mssge_ss($pckt.type, $pckt.data[])
            charpos = 1
            numpckts = 0
            CALL disassemb_real(theta, $pckt.data[], charpos, numpckts)
            TYPE
            TYPE "x= ", deltax, "  y= ", deltay, "  theta= ", theta
            CALL recv_mssge_ss($pckt.type, $pckt.data[])
            charpos = 1
            numpckts = 0
            CALL disassemb_real(col_framex, $pckt.data[], charpos, numpckts)
            CALL recv_mssge_ss($pckt.type, $pckt.data[])
            charpos = 1
            numpckts = 0
            CALL disassemb_real(col_framey, $pckt.data[], charpos, numpckts)
            TYPE
            TYPE "col_frame.x = ", col_framex, "  col_frame.y = ", col_framey
            TYPE
            CALL recv_mssge_ss($pckt.type, $pckt.data[])
            charpos = 1
            numpckts = 0
            CALL disassemb_int(flag, $pckt.data[], charpos, numpckts)
            IF flag THEN
;
; assemble robot trajectory from error vector
;
              BREAK
              SET alt14[1] = TRANS(col_framex+6,col_framey-763.304)
              SET alt14[2] = TRANS(,,,,,-theta)
              SET alt14[3] = TRANS(-(col_framex+6),763.304-col_framey)
              SET alt14[4] = TRANS(,,,,,theta)
              SET rotate14 = alt13:alt14[1]:alt14[2]:alt14[3]
              SET alt15 = TRANS(deltax,-(deltay+55),111.265) ;
              SET cart15 = rotate14:alt14[4]:alt15:alt14[2]
              SET alt16 = TRANS(,25,56) ;
              SET cart16 = cart15:alt16
              SET alt17 = TRANS(,10,-9)
              SET cart17 = cart16:alt17
```

```
            DECOMPOSE decomp17[] = cart17
            TYPE "cart17:  x= ", decomp17[0]
            TYPE "         y= ", decomp17[1]
            TYPE "     theta= ", decomp17[5]

            TYPE "Elected modified locations"
;
; synchronize robot movements with SS system commands
;
            CALL signal_ss() ; rc 010
            SPEED 50 ALWAYS
            MOVE alt13
            BREAK
            SPEED 5
            MOVE rotate14
            BREAK
            SPEED 10
            MOVE cart15
            BREAK
            CALL signal_ss() ; rc 011
; turn on vacuum
            CALL signal_ss() ; rc 012
            SPEED 15
            MOVE cart16
            BREAK
            MOVE cart17
            BREAK
            CALL signal_ss() ; rc 013
         ELSE
            TYPE "Elected preprogrammed locations"
            CALL signal_ss() ; rc 010
            SPEED 50 ALWAYS
            MOVE loc[13]
            BREAK
            MOVE loc[14]
            BREAK
            MOVE loc[15]
            BREAK
            CALL signal_ss() ; rc 011
            CALL signal_ss() ; rc 012
            SPEED 15
            MOVE loc[16]
            BREAK
            MOVE loc[17]
            BREAK
            CALL signal_ss() ; rc 013
         END
         CALL signal_ss() ; rc 014
         MOVE loc[18]
         BREAK
         MOVE loc[19]
         BREAK
         CALL signal_ss() ; rc 015
         CALL signal_ss() ; rc 016
         MOVE loc[0]
         BREAK
;******************
;*              *
;*  END OF PLAN.  *
;*              *
;******************
         TYPE "Program Completed!"
         WHILE TRUE DO
         END
.END
```

LIST OF REFERENCES

1. Ishadawa, S., "On the Large-Scale Project: "Automated Sewing System"," JIAM, Japan 1990.

2. Gershon, D. and I. Porat, "Vision Servo Control of a Robotic Sewing System," Proceedings of the IEEE International Conference on Robotics and Automation, Philadelphia, April 1988, pp. 1830-1835.

3. Taylor, P.M., A.J. Wilkinson, G.E. Taylor, M.B. Gunner, G.S. Palmer, "Automated Fabric Handling Problems and Techniques," IEEE International Conference on Systems Engineering, Pittsburgh, 1990, pp. 367-370.

4. Norton-Wayne, L., "Automated Garment Inspection Using Machine Vision," IEEE International Conference on Systems Engineering, Pittsburgh, 1990, pp. 374-377.

5. Taylor, P.M., A.J. Wilkinson, I. Gibson, M.B. Gunner, G.S. Palmer, "An Integrated Automated Garment Assembly System," IEEE International Conference on Systems Engineering, Pittsburgh, 1990, pp. 383-386.

6. Kelley, R.B., "2-D Vision Techniques for the Handling of Limp Materials," NATO ASI Series: Sensory Robotics for the Handling of Limp Materials, Vol. F64, Springer-Verlag Berlin, 1990, pp. 141-157.

7. Iype, C. and I. Porat, "Fabric Alignment Using A Robotic Vision System," International Journal of Clothing Science Technology, Vol.1.1, 1989, pp. 39-43.

8. Torgerson, E. and F.W. Paul, "Vision Guided Robotic Fabric Manipulation for Apparel Manufacturing," IEEE International Conference on Robotics and Automation, Raleigh, 1987, pp. 1196-1202.

9. Ogawa, S., "R&D in Automated Sewing System," JIAM, Japan, 1990.

10. Halioua, M. and H. Liu, "Optical Three-Dimensional Sensing by Phase Measuring Profilometry," Optics and Lasers in Engineering, Vol. 11, 1989, pp. 185-215.

11. Sato, K. and S. Inokuchi, "Range-Imaging System Utilizing Nematic Liquid Crystal Mask," Proceedings of the IEEE First International Conference on Computer Vision, London, June 1987, pp. 657-661.

12. Sato, K. and S. Inokuchi, "Three-Dimensional Surface Measurement by Space Encoding Range Imaging," Journal of Robotic Systems, Vol. 2, No. 1, 1985, pp. 27-39.

13. Elrom, I. and A. Hermala, "Intelligent Vision Based Contour Sensor Update," ISA Calgary '89 Symposium, April 3-5, 1989, pp. 359-368.

14. Kehtarnavaz, N., "A Syntactic/Semantic Technique for Surface Reconstruction from Cross Sectional Contours," Computer Vision, Graphics, and Image Processing, Vol. 42, 1988, pp. 399-409.

15. Domey, J., M. Rioux, and F. Blais, "3-D Sensing for Robot Vision," NATA ASI Series: Sensory Robotics for the Handling of Limp Materials, Vol. F64, Springer-Verlag Berlin, 1990, pp. 167-202.

16. Paul, F.W., E. Torgerson, S. Avigdor, D.R. Cultice, A. Gopalswamy, and K. Subba-Rao, "A Hierarchical System for Robot-Assisted Shirt Collar Processing," IEEE International Conference on Systems Engineering, Pittsburgh, 1990, pp. 378-382.

17. Torgerson, E.J., A Heirarchical Semi-Autonomous Control Scheme for Robot -Assisted Workstations, PhD. Thesis, Mechanical Engineering Department, Clemson University, Clemson, S.C., December 1991, Chap. 5.

18. Figliola, R.S., and D.E. Beasley, Theory and Design for Mechanical Measurements, John Wiley & Sons, Inc. New York, 1991, Chap. 5.

19. Luh, J.Y.S., and J.A. Klaasen, "A Three-Dimensional Vision by Off-Shelf System with Multi-Cameras," IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-7, No. 1, January 1985.

20. Castleman, K.R., Digital Image Processing, Prentice Hall, Englewood Cliffs, New Jersey, 1979, Chap. 17.

21. Subba-Rao, K., Vision-Assisted Edge Alignment for the Automated Pressing of Two-Dimensional Apparel Components, MS. Thesis, Mechanical Engineering Department, Clemson University, Clemson, S.C., August 1991.

22. Gopalswamy, A., Design and Control of a Robot End-Effector for Three Dimensional Manipulation of Multiple-Ply Apparel Workpieces, MS. Thesis, Mechanical Engineering Department, Clemson University, Clemson, S.C., August 1990.

23. Nakamura, T., T. Arai, Y. Tanaka, M. Satoh, and Y. Imazu, "Trajectory Generation of Redundant Manipulator for 3-D Sewing System," ASME Proceedings on Flexible Automation, Minneapolis, MN, July 1988, pp. 35-40.